2012-05-23

Layout Optimization for Distributed Relational Databases Using Machine Learning

Jozsef Patvarczki
Worcester Polytechnic Institute

Layout Optimization for Distributed Relational Databases Using Machine Learning

By

Jozsef Patvarczki

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

_____

May 2012

APPROVED:

_____	_____
Dr. Neil T. Heffernan Advisor	Dr. Craig E. Wills Committee Member Head of Department
_____	_____
Dr. Elke A. Rundensteiner Committee Member	Dr. Daniel C. Zilio IBM - Toronto External Committee Member

ABSTRACT

A common problem when running Web-based applications is how to scale-up the database. The solution to this problem usually involves having a smart Database Administrator determine how to spread the database tables out amongst computers that will work in parallel. Laying out database tables across multiple machines so they can act together as a single efficient database is hard. Automated methods are needed to help eliminate the time required for database administrators to create optimal configurations. There are four operators that we consider that can create a search space of possible database layouts: 1) denormalizing, 2) horizontally partitioning, 3) vertically partitioning, and 4) fully replicating. Textbooks offer general advice that is useful for dealing with extreme cases - for instance you should fully replicate a table if the level of insert to selects is close to zero. But even this seemingly obvious statement is not necessarily one that will lead to a speed up once you take into account that some nodes might be a bottle neck. There can be complex interactions between the 4 different operators which make it even more difficult to predict what the best thing to do is.

Instead of using best practices to do database layout, we need a system that collects empirical data on when these 4 different operators are effective. We have implemented a state based search technique to try different operators, and then we used the empirically measured data to see if any speed up occurred. We recognized that the costs of creating the physical database layout are potentially large, but it is necessary since we want to know the "Ground Truth" about what is effective and under what conditions. After creating a dataset where these four different operators have been applied to make different databases, we can employ machine learning to induce rules to

help govern the physical design of the database across an arbitrary number of computer nodes. This learning process, in turn, would allow the database placement algorithm to get better over time as it trains over a set of examples. What this algorithm calls for is that it will try to learn 1) "What is a good database layout for a particular application given a query workload?" and 2) "Can this algorithm automatically improve itself in making recommendations by using machine learned rules to try to generalize when it makes sense to apply each of these operators?"

There has been considerable research done in parallelizing databases where large amounts of data are shipped from one node to another to answer a single query. Sometimes the costs of shipping the data back and forth might be high, so in this work we assume that it might be more efficient to create a database layout where each query can be answered by a single node. To make this assumption requires that all the incoming query templates are known beforehand. This requirement can easily be satisfied in the case of a Web-based application due to the characteristic that users typically interact with the system through a web interface such as web forms. In this case, unseen queries are not necessarily answerable, without first possibly reconstructing the data on a single machine. Prior knowledge of these exact query templates allows us to select the best possible database table placements across multiple nodes. But in the case of trying to improve the efficiency of a Web-based application, a web site provider might feel that they are willing to suffer the inconvenience of not being able to answer an arbitrary query, if they are in turn provided with a system that runs more efficiently.

Acknowledgements

I would like to say thank you to many people who helped me to complete my dissertation and I would like to acknowledge them in the completion of this dissertation. First of all, I want to thank my advisor Neil T. Heffernan for his extraordinarily support and for his time and patience advising me. His excellent advice and continuous energy inspired me to reach my goals.

Secondly, I would also like to thank my dissertation committee, Elke A. Rundensteiner, Craig A. Wills and Daniel C. Zilio. Their support and professional expertise were always available during my work.

Thirdly, I would like to say thank you to Gabor Sarkozy for all of his consistent help and encouragement at WPI and for his great help on the math chapter.

The ASSISTments team is an extremely talented group of people. I was one of the luckiest candidates who had the chance to work with them. Cristina Heffernan, Zach Pardos, David Magid, and so many others have been colleagues.

Finally, thanks to all of my family members who supported me. Special thanks to my wife Aniko for her love and patience during my studies. Also, infinite thanks to mom and dad for encouraging me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

1.1. Motivation

Nowadays, Database Administrators have to be familiar with multiple available data stores to select the best fit for their Web-based applications. Upon a successful selection, scalability issues related to the database could be a possible bottleneck of the system. Especially, if the requests are distributed among multiple database servers that could lead to slow response time or a possible system crash. Scalability issues and their possible solutions should be automatically addressed without spending an enormous amount of time on investigating the database structure or investing into expensive hardware solutions.

We propose a rule-based distributed database design framework that has a novel assumption: we can increase the total throughput of a Web-based application by automatically creating database configurations that are capable of answering each of the incoming query templates using a single node. For example, if Q1: ”SELECT * from T1” and Q2: “SELECT * from T1 WHERE id=12” are two different query templates and table T1 is partitioned based on key “id”, then we can answer Q1 template using a single node because it has the right partitioning key in the “WHERE” clause. We conceptualize the database layout problem as a state space search problem. A state is a given assignment of tables to computer servers. We begin with a database and collect, for use as a workload input, a sequence of queries that were executed during normal usage of the database. The operators in the search are to fully replicate, horizontally partition, vertically partition, and denormalize a table. We do a time intensive search over different

table layouts, and at each iteration, physically create the configurations, and evaluate the total throughput of the system. To make this search more practical, we want to learn reasonable rules to guide the search to eliminate many layout configurations that are not likely to succeed. There can be complex interactions between the four different operators which make it even more difficult to predict what the best way to do is. After collecting empirical data, we use the created configurations as input into a machine learning component, to predict when to use the different layout operators and to induce rules to help govern the physical design of the database across an arbitrary number of computer nodes.

1.2. The System

This research was carried out with our implemented framework [42], a middleware architecture that is based on shared-nothing commodity hardware where each node has its own CPU, disk, RAM, and file system. We focused on a Web-based application where the workload consists of a fixed number of query templates. This means the system is not presented with ad-hoc and unexpected queries. According to our best knowledge, none of the existing systems specialize for Web-based applications, utilizing machine learned rules and consider the database layout problem as a state space search problem with the assumption that all the incoming queries should be answered by a single node. By characterizing the problem as a state space search over database layout configurations, the system iteratively minimizes the total cost of the workload creating different database layouts and increasing the total system throughput.

TPC-W [43], the Industry Standard eBusiness transactional web benchmark's tables and query templates were used to generate different databases and to measure the response time of the implemented system. Its workload simulates the activities of a retail store website.

1.3. Research Questions

This dissertation attempts to answer the following questions:

1. What is a good database layout for a particular Web-based application given a query workload?

2. Can our layout algorithm automatically make recommendations by using machine learning technique to try to generalize when it makes sense to apply each of these operators?

3. Can we learn rules that are effective at speeding up the whole system?

- How can we parameterize these rules for cut-off values?
- What are the possible sets of important features that we need to take into consideration to learn a general rule?

By rules we mean like:

- "*when the number of update/delete/insert queries on a table are small compared to the number of retrieval queries (e.g selects), then one should fully replicate*";
- "*if there is a wide table but a lot of read queries focused on a small set of columns*

of the table, then one should vertically partition";

- "*when the table size is large, then one should horizontally partition*"

4. If these rules are effective in general, then does this system become more efficient with laying out databases over time?

- Can we make the search for layout for a new system more efficient over time? If the rules we learned are good, then we could use the rules themselves to bias a search for layout for a new database.

1.4. Contributions

This dissertation makes four main contributions to the field of database design in computer science. The first contribution is a layout algorithm that is capable of determining a possible data placement based on the query templates, constraints, and the optimization goal using four operators (full replication, horizontal partitioning, vertical partitioning, denormalization) and arbitrary number of database servers answering each query by a single node.

The second contribution concerns designing and developing a shared-nothing data replication framework for Web-based applications with state based search and machine learning components to predict when to choose between horizontal partitioning, vertical partitioning, denormalization or full replication layout operators. We collected empirical data that reflect trade-off values of the best practices to help the state space search to focus on creating layout configurations that could boost the performance of the

application. These data are the key to know the "ground truth" about what is effective and under what conditions. After collecting empirical data, where these four different operators have been applied, we used the created configurations as input into a machine learning technique.

The third contribution is the machine learned rules to help govern the physical design of the database across an arbitrary number of computer nodes. These rules, in turn, allow the database placement algorithm to get better over time as it trains over a set of examples.

Our fourth contribution is a comparative analysis of the trade-off values to be able to assign confidence values to each operator and determine their precedence as a "molecular structure".

1.4.1. Main Assumptions

This thesis makes three fundamental assumptions:

1) We know ahead of time every query template that could come to the system

We focused on a Web-based application where the workload consists of a fixed number of query templates. This means the system is not presented with ad-hoc and unexpected queries. A characteristic of a web application such as Amazon (www.amazon.com), is that we know all the incoming query templates beforehand as the users typically interact with the system through a web interface [1]. The application logic executes the same hard-wired queries over and over again for the same web form request.

2) We demand that each query will be answered by a single database node

The techniques described in this dissertation assume that the data is distributed on different database servers in such a manner that the requested information is answered by one database server. There is significant work that has studied scenarios without this constraint. Distributed databases and distributed query processing [49] have long studied how to process queries over data distributed across multiple nodes. However the constraint that any select query is answered by one database server is applicable to several applications, especially web applications where all the query templates are known beforehand (assumption 1). This constraint also greatly simplifies query processing and optimization, as no data needs to be exchanged between nodes. Therefore such a system has to only determine which database server needs to execute a query, and then the optimization and execution of the query proceeds on that server as if it were a non-distributed database.

3) Initial Data Distribution Policy: Tables will be fully replicated across all nodes

As the start condition we fully replicate all tables across all database nodes and measure the response time of the system using the given workload. We compare all further measurements and layouts to this start distribution policy. This step is necessary to guarantee that we will always fulfill the second assumption by utilizing the benefits of the distributed infrastructure and by preventing the system from partitioning all tables under a single node.

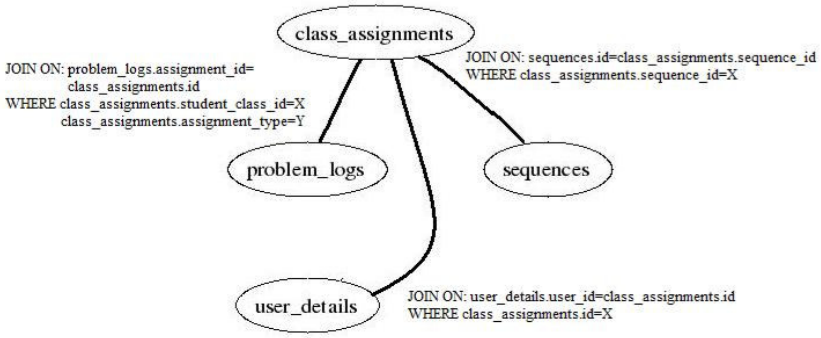


Figure 1: The join-graph and the partitions

An SQL join clause queries data from two or more tables, based upon the relationship between columns in the involved tables e.g. "SELECT * from user_details INNER JOIN class_assignments ON user_details.user_id = class_assignments.id WHERE class_assignments.id = X". For example, the edge between user_details and class_assignments indicates the join relationship between the two tables (see Figure 1). The join-graph is capable of identifying a group of tables as individual join partitions. Figure 1 shows one join-graph partition where class_assignment, problem_logs, sequences, and user_details are involved in joins. Each node of the graph is a table and each edge is a join relationship. Most of the applications have more than one join-graph partitions but there is no guarantee for that. If the application is highly coupled (has only one big partition), then the likelihood that one can find a single component, further partition that into smaller ones, and distribute the data across multiple nodes is small. According to our case-studies, most of the Web-based applications have one big partition and a couple of small individual ones. Appendix A shows our case-study results.

1.4.2. Minimizing the Response Time of a Web-based Application

There are thousands of web applications (Amazon, eBay, etc.), and these systems need to figure out how to scale-up their performance. Distributing load across multiple application servers is fairly straightforward but distributing load (read, update, delete, and insert queries) across multiple database servers is more complex. Data partitioning is a time-consuming heavily used method for minimizing the response time of a Web-based application. Database Administrators (DBA) have to spend enormous time and energy to analyze data and come up with the right solutions to support more clients with adequate service response time. DBAs want to know how to adapt their systems to scale-up and achieve the best results. Researchers of cloud computing also want to understand how users interact with a Web-based system and how to generate database intensive input information. One of the applicable solutions is to build an expensive storage network that is capable of scaling up the backend. Unfortunately, this network does not guarantee that the backend of our Web-based application can be scaled-up to an expected limit. Moreover, the hardware cost of this setting can be high. It would be nice to know the application's scaling possibilities beforehand and do the hardware investment thereafter. The second applicable but possible painful solutions is to abandon the relational models and port the SQL-based systems to a NoSQL database service and use one of the hybrid solutions [44][45][46][47]. This solution could lead to improvement but to be able to proceed DBAs have to have deep knowledge about data and information architecture of their system. Also, most of the NoSQL services do not support specific SQL keywords (e.g. join) or indexing strategies, and therefore the application code must be changed as well.

Choosing a good physical database design, using different operators, is an essential task for almost any automatic physical database design tools. Robust physical database design can have a great impact on the system performance. Our solution gives a simple tool to the developers of Web-based technologies with relational databases. It can be easily built using low-cost existing resources to realize database-scaling possibilities without the necessary application porting or expensive storage area networks. Automatic physical database design tools mostly rely on "what-if" [9] that estimates the execution time of the queries and recommends adjustment of the layouts involving one of the partitioning operators based on that. Researchers in database technologies are interested in methods for exploiting system characteristics. A Microsoft Research paper "Query Optimizers: Time to Rethink the Contract?" [48] suggests revisiting the contract for query optimizers and to be able to gather additional information to achieve scalability from usage-based analysis e.g. search directives. Our framework applies state based search over database layouts and combines different partitioning guidelines utilizing full-replication, horizontal and vertical partitioning, and denormalization operators.

1.4.3. State Based Search over Database Layouts

Database designs have been studied in the past [69, 70, 71, 72, 73, 74, 75]. By characterizing the problem as a state space search over database layout configurations, we iteratively minimize the total cost of the workload creating different database layouts and increase the total system throughput. Our optimization goal is to minimize the total system response time by figuring out how to best distribute the data. We do a time intensive search over different layouts, and each time, physically create the

configurations, and evaluate the response time of the system. A state is a given assignment of tables to computer servers and it represents the actual status of the system. The operators in the search are to fully replicate, horizontally partition, vertically partition, and denormalize a table. After each valid state creation the system measures the total response time of the system using the query workload to get actual performance measures from a real setup. Through our experiments and construction we used these real numbers in place of a possible estimator to demonstrate our algorithm's functionality. The search over layouts can be very expensive and a possible virtual partitioning (Chapter 6.4) or a DBMS optimizer (like IBM DB2 [10]) can be used to predict numbers as a replacement black-box component for actual performance measures. However, entities trying to scale up their Web-based applications would be perfectly happy to prefer real run-time measurements over estimated ones and spend a few weeks of CPU time to increase their system throughput.

We apply search algorithm where the idea is to always move towards a state that is better than the current one. If the state is not better than the current one (the total system response time is worse) we do not continue the search along that path. The search terminates when no more states are left to be explored. We understand that heuristics searches converge to local minimums but this convergence is an acceptable compromise to achieve a faster search.

1.4.4. Machine Learned Rules

To reduce the time intensive search over layouts and to make this search more practical, we apply our reasonable determined rules to guide the search to eliminate many layout

configurations that are not likely to succeed. There can be complex interactions between the four different operators, which make it even more difficult to predict what the best way to do is. After collecting empirical data where these four different layout operators have been applied, we use the created configurations as input into a machine-learning component, to predict when to use them. This learning, in turn, would allow the database placement algorithm to get better over time and reduce the execution time of a long running brute-force search.

1.5. Dissertation Outline

The remainder of this dissertation is outlined as follows. Chapter 2 presents prior work on different frameworks: industry, academia research, and hybrid solutions. Chapter 3 describes the problem statement, state space search, data placement algorithm, and partitioning methods. It introduces us to the world of possets and their graphical representations. Chapter 4 introduces the implemented framework and its components. It shows the complete workflow of the framework combined with performance evaluation using TPC-W industrial benchmark. Chapter 5 shows how we can determine trade-offs based on database best practices and presents the collected empirical data. Chapter 6 introduces our machine learned rules, features, matrix models, ground truth, and the analysis of the model predictions. Chapter 7 performs a comparative analysis on the determined rules and it discusses a method how to vote on the best operator. It also connects the learned rules with best practices and shows how to create a representation of the operator precedence as a “molecular structure”.

2. Related Work

This chapter describes prior work on industry and academia research. It also discusses hybrid data store solutions.

2.1. Overview

Scaling up web applications requires distribution of load across multiple application servers and across multiple database servers. Distributing load across multiple application servers is fairly straightforward; however distributing load (select and UDI queries) across multiple database servers is more complex because of the synchronization requirements for multiple copies of the data.

There are thousands of web applications, and these systems need to figure out how to scale up their performance [18]. Issues related to the distribution of requests among multiple database servers to decrease database server loads have stayed open. A characteristic of web applications is that all the incoming query templates are known beforehand as the users typically interact with the system through a web interface such as web forms [1]. Others have already tried to take advantage of this fact [1][17][14]. Knowing each query template in advance allows us to propose better solutions for balancing load across multiple servers in the scenario of web applications, above and beyond what is supported for traditional applications. Prior knowledge of all of the incoming query templates and the workload give us the ability to select an appropriate table placement where each query template can be answered with a single database server.

2.2. Review of Industry Research

While there has been work in the area of automating physical database design [7] [8] we are not aware of any work that addresses the problem of incorporating the full range of common operators (full replication, horizontal and vertical partitioning, and denormalization), and can learn rules to better partition a given database with multiple database nodes.

For example, in [6] they automatically select an appropriate set of materialized views and indexes to optimize the physical design for a given database and query workload as a built in tool in Microsoft SQL server 2000 using a single node. Their system has a candidate selection module that identifies the set of indexes and materialized views for the given workload that are worth consideration. They do not make our key assumption about the known query templates but their candidate selection module tries to reduce the layout possibilities. They suggest developing a physical database design tool that is able to apply more data operators (e.g. vertical partitioning and de- normalization) to achieve better system performance.

The paper [6] has tackled the related problem of how to figure out how to automatically know which indexes to put on the system and which materialized views apply. This is similar to our approach where we try to use horizontal and vertical partitioning, full replication, and denormalization operators to lay out tables. They, like our approach, assume they know the workload. Unlike our approach, they only deal with a single database while we deal with multiple bases. In their solution they have more assumptions, and they try to figure out which materialized views to build, which is a

topic we do not address. We should be able to take advantage of the work in [6] to apply it as a post processing step after our algorithm is complete.

In paper [9], they added a new operator, called horizontal partitioning, to the previous optimization goal in Microsoft SQL server 2005. Microsoft SQL server 2005 offers an automated database design tool that has physical design recommendation for horizontal partitioning. It recommends to give ability to database administrators to specify alignment requirements for data partitioning while optimizing for performance. Compared to our multiple database node handling, their system handles databases on a single node. Their output is a physical design recommendation on horizontal partitioning of tables, materialized views, and indexes. They have a query optimizer that has a cost model ("what-if") for queries. Their system does not assume that they know all the query templates beforehand. Instead, they try to filter the workload and capture parameters like which column group has higher impact on the workload, and fine tune with workload compression. They realized that knowing about the query templates can lead to a better optimization result. They assume that queries in the workload often belong to the same template and this fact can help to tune their system with workload compression. This workload compression captures the similar queries and tries to determine the different query templates. Compared to our automated data replication middleware, database administrators have to copy the original database to a test server and run the advisor on the dataset. After this step, the system will recommend a physical design to achieve the best performance and the administrator has to configure the original database manually.

IBM DB2 Design Advisor [10] is a tool, that for a given workload, automatically recommends physical design features for indexes, materialized views, horizontal and

vertical partitioning for a single-node. We are not considering indexing in our system but we could use the idea from [10] as a preposition step. This tool [10] is the first one that supports several operators. It has a workload compression module that reduces the workload size automatically. In their experience, the workload was not given and they realized it is really hard to collect information about infrequent queries. Therefore, they created a workload compression module where an administrator can give a workload from command line, file, or an existing workload table. To increase the efficiency of the workload they "take an approach that only keeps the top K most expensive queries, whose total cost is no more than X% of the original workload cost". After this approach, they will sort the statements in descending order combining with continuous selection from the top to increase the efficiency of the workload. As a final outcome, their system displays the results and their initial result shows that the design advisor can improve the performance of workloads. They define dependencies among operators as strong or weak one to classify them for easier decision. They introduce iterative and integrated approach for layout design. In the first case, the interaction between each operator is ignored and it can handle each operator selection as a black box. Iterative approach searches the entire search space of the combined operators. They determined the dependencies between each operator. Their solution is a hybrid one that combines the two approaches (iterative and integrated) into one. Unlike our work, where we have working code for horizontal and vertical partitioning, full replication, denormalization and that not only make a suggestion, we actually implement the suggestion by doing the database layout. There are features they don’t do in this paper that we seek to do. First of all, the system attempts to give "advice" but it does not actually implement the advice. For example, it might

suggest to horizontally partition a table, but it won't do it for you, nor will it give you a query router that would handle the queries coming into the system. Probably more importantly, since the system does not implement its advice automatically, it can't check to see if the advice really is going to result in an increased performance on the workload that is given. They assume that the "subsequent searching only further improves the performance marginally". This assumption is their stopping criteria for their algorithm. They will stop the advisor when no further improvement is possible after a certain number of iterations. According to their results, they implemented the advisor recommendations and measured the actual cost of the workload for the TPC-H benchmark. The achieved real performance improvement as 84.54%, which is close to the advisor's estimation (88.01%).

There is an empirical question about whether the system can apply all the operators in a single pass, or whether it is really worthwhile to do it one step at a time (multi pass). The later procedure will consume more CPU time but might give better performance result. The single pass execution time could be shorter but might not give as good of a performance result as the multi pass. Our system considers the multi pass procedure as default but it could apply the single pass as an optional configurable parameter. Furthermore, we can use [10] as a pre-processing step to figure out what indexes do we need to build, and assume that those indexes will be worthwhile to keep even after the parallelization step we propose.

While in [10] they focused on a single node, in the next paper [7], IBM DB2 looked at the problem of laying out multiple nodes using many operators like we propose. [7] is certainly the work that most closely relates to our work. In [7] they are concerned

with our same problem in the IBM DB2 Enterprise Extended Edition. They are similar in that they consider horizontal partitions and full replication and they are different because their system also considers materialized views. Compared to our middleware, their system is based on a shared-nothing architecture where a collection of nodes is used for parallel query execution. This means that their tool uses multiple nodes for partitioning tables horizontally. They do not have the assumption that each query needs to be answered using a single node. They assume that the workload of SQL statements is given, as assumed by our system. Their optimization goal is to achieve optimal performance of the given workload. Their tool suggests possible partitions based on the given workload and the frequency of each SQL statement occurrence. Their approach "only recommends one best candidate partition for each table referenced by the query to determine good candidate partitions for each table in each individual statement". Furthermore, their system recommends only the best operator (partitioning or full replication) for each table. They introduced "RECOMMEND" mode to find the optimal partition of a table for each query. They compute a set of interesting candidates that can help to reduce the query cost and storing them in the "CANDIDATE_PARTITION" table. For example, they consider the size of the table to decide about replication. They will replicate the table if the size is smaller than a threshold. Because we know the incoming query templates we can replicate the table if the number of SQL selects are significantly higher than the number of UDIs. They consider materialized views as an option to improve performance (e.g. to handle joins across nodes) which is a topic we do not address. They have an interesting approach for cost estimations of the queries. Their overall cost is a combination of different constraints (I/O, CPU, and communication

cost). They have an important assumption: "repartitioning is an expensive process and is not expected to be run frequently". We share this assumption especially if we know all the incoming query templates then it is not necessary to redesign the complete structure upon a possible new template (if the workload characteristics or the data distribution have not changed significantly). We just have to update the router logic with the new template. If they will create a structure then the DB2's built in router logic is responsible for maintaining it and answering each query using multiple nodes.

Paper [8] describes IBM DB2 partitioning selection strategy for a given static database schema and workload characteristic to minimize the overall response time of the workload in a multiple node environment. In [8] they introduced the function-shipping model that manages IBM DB2 query execution. This model minimizes the intercommunication cost between nodes with answering the queries using a single node (e.g. joins are done locally). It is really important that the data placement can recognize the partition specific attributes of the templates. They introduce two main placement algorithms the Independent relations and the Comb one. The Independent Relation considers each query attributes separately. For example, if there is a query 'select book from table1, table2 where table1.book_id = table2.book_id', then the algorithm considers table1.book_id as a partition key first and then table2.book_id. The Comb algorithm considers the combination of the keys, e.g. table1.book_id and table2.book_id together. After the partition key decision, the algorithm groups the relations together and determines the node to which to assign the partition key to be based on their relation grouping technique. The created groups contain different tables, e.g. group1 contains t1, and group2 contains t2 and t3. If the partitioning key is book_id for each, then t2.book_id

is guaranteed to be on the same node as t3.book_id. Compared to our assumption that we know all incoming query templates and the queries will be answered by a single node we guarantee that t1.book_id , t2.book_id, and t3.book_id will be on the same node and that they will belong to the same group.

2.3. Review of Academia Research

GlobeTP [1] exploits the same fact that our system does: the workload of the web application is composed of a small set of query templates. They predict query execution costs based on the known templates and using the result for table placement involving table replications, they can only answer the queries they are prepared for. They employ full replication and use a replication-like operator to improve the total throughput. Their replication like operator replicates the entire table on a sub-set of database nodes. In the database domain, partial replication assumes that shared data is partitioned into *n* disjoint databases and we allow replication of an arbitrary subset of the databases as long as every database is present on at least one node. They are similar in that they make the same assumption that our system uses: each query template can be treated locally by at least one server. This means that there is at least one server that is able to execute each query. Their system is different because the only operator that they use is replication. Each server has a replica of one or more tables of the original database. Their main goals are to increase the total system throughput and decrease the query access latency. They have a query router that routes the incoming query to the appropriate node that contains all tables to answer the request. The router knows the current placement of the tables and it is responsible for maintaining the consistency of the system. If a UDI query comes in,

then the router will execute it on all the servers that hold the required tables. The router serializes the incoming queries to maintain consistency and it handles each read or write independently. Their query router has two main routing policies: the round-robin per Query ID and the Cost-Based Routing. In the first case, each query template has its own queue and each queue has a set of databases that can answer the template related queries. In the second case, the router can estimate the load on each database server and routes the queries to the least loaded database server that has the required set of tables to answer the request. Our system is similar because it has router logic too. But in our case each database server is managed by a thread that maintains two data structures: a queue of requests it has received, and a lock table to handle conflicting select and UDI queries. In order to increase the performance of each database server, the thread for the database server maintains multiple connections to that server; thus multiple queries can be executed simultaneously on a single server to minimize the average response per query. Their system is efficient if the application has few UDIs compared to selects, but to schedule and maintain consistency when the number of UDIs is high is a real bottleneck, especially if the system has large number of database nodes with fully replicated tables. One of the obvious drawbacks is if there is a table with a couple million rows then their replication operator distributes the same data on multiple nodes creating high storage cost.

In [17] their approach describes two common properties of web based applications. According to their strong assumption, workload is dominated by reads and it consists of a small number of query and update templates (typically between 10 and 100). Using the second assumption, their system solves strong consistency management of the

servers. They use a fully distributed consistency mechanism that leverages the fact that querying and updating is mainly restricted by the templates specified in advance. This assumption is similar to our system because they realized if the query templates are known beforehand then their consistency mechanism can achieve better performance but their system does not know the total workload beforehand. They have proxy servers for replicating the query results with distributed consistency management that has an efficient routing mechanism for messages. The proxy server has a copy of the web server and the application server, and acts as a database containing read-only copy of the query results. Their infrastructure has a home server with a master copy of the database at the back-end. If there is an insert, update, or delete they have to distribute the request among all of the proxy servers and invalidate the old data in the caches. They adopt a simple consistency model and group the query templates into cacheable and uncacheable ones to help reduce the update cost. Clearly their system is similar to ours because it tries to use a single proxy node to answer the query before connecting to the home server. Moreover, they want to build a full replica of the known query results dynamically and distribute it with proxy servers. The drawback of this approach is if the system has many proxy servers then to keep maintaining the consistency is inefficient (query caching requires high temporal locality) and the system throughput can be limited by them.

DBProxy [14] observed that most applications issue template-based queries and these queries have the same structure that contains different strings or numeric constraints. This observation helps to reduce containment checking overhead significantly. Their research - similarly to ours - assumes they know the workload and they can pick a good caching strategy. Their system is a semantic data cache designed for

adopting changes in the workload. They aggregate the similar query templates in the cache, which leads to a faster query search and significant performance improvement. Their system caches the materialized views of the given workload. This method uses different tables for views that can be set by the administrator. They have a query evaluator and a caching logic decides which query result should be cached. The query-matching module checks the query templates and directs them according to the views. Their system benefits from the known query templates -similarly to our system - as a template-based matching of the queries. Their assumption is that the system can reduce the containment checking by a significant amount. Queries belonging to the same template are aggregated and can help to identify the query and get the result much faster. They realized that their solution suffers from the same problem that [7] suffers, so when one of the tables gets hit by heavy UDIs they will disable the copies for a specified time period. With his solution they propagate UDIs directly to the main database and later the data will be updated to the database caches by the data propagator module.

AutoPart [19] deals with large scientific databases where the continuous insertions limit the application of indexes and materialized views. For optimization purposes, their algorithm horizontally and vertically partitions the tables in the original large database according to a representative workload using a single node. Their solution is similar to our system that considers these two operators as well. They do not know the query templates beforehand but their system has a Query Access Set component to capture each query access frequency and determine their cost with a query optimizer of a database system. Their result shows that the partitioned schema can speed up the query execution time without indexes and the new schema even performs better when indexes

are applied after the partitioning phase and not on the original database. They expect to lower the workload cost with the partitioned design, because it is faster to access the partitions and queries do not need to access useless attributes. Their optimization goal is to increase the query execution performance with partitioning and recommend a physical design using a single node. Compared to our system their work assumes a given query router logic and a consistency management solution.

Paper [15], an addition to [6], talks about the importance of horizontal and vertical partitioning operators for physical design in relational databases using a single node. These operators can significantly impact the performance of the workload. They assume that combining these operators with alignment requirements can lead to significant performance improvements. They consider layouts where the structures on each table are aligned (identically partitioned). This paper is similar to our system because they consider partitioning operators and they try to combine these operators with alignment and manageability assuming that the workload is given. A big difference is that their system focuses on the single-node partitioning where all objects are presented on a single server and they do not consider the incoming query templates to be known. The paper introduces alignment and it considers indexes aligned if they are horizontally partitioned in the same way as the related tables. Tables can be partitioned differently according to the different queries and the partitioning requirements. They divide their horizontal partitioning operator into range and hash partitioning. Their optimization goal is to optimize the database for a given workload to decrease the query access time. They model the workload as a set of SQL statements and with each statement they associate a weight to capture the multiplicity of a given SQL statement in the workload similarly to

our system. They determine the cost of each statement using "what if" or assume that is given by the database optimizer.

In [5], their optimization goal is to restructure data services into multiple independent ones with separated data access using a denormalization operator on select queries. Denormalization takes the benefit of special queries and transactions which often access only a part of the columns of a table. One can decompose such tables into multiple ones to simplify the workload and to optimize the efficiency of query execution. These restructured data structures can lead to a total throughput improvement using multiple nodes. In denormalization, one moves from higher to lower normal forms in the database modeling and adds redundant data. The performance improvement is achieved because some joins are already pre-computed. However there are disadvantages. For instance UDI queries are cumbersome when performed against denormalized data, as we need to synchronize between duplicates. This paper [5] applies denormalization on web services directly to distribute them with a possible caching solution. If the application has a large number of query templates an appropriate caching mechanism can also help to scale as an additional technique to data placement. Instead of using a single database node, their system splits the application data into three databases. Each database is encapsulated into a data service that creates a bridge to the business logic. According to the access patterns of the databases, each data service and its database can be further divided into databases on different nodes and different operators can be applied on them. For example, the first denormalized database can be partitioned into two databases and the second database can be replicated across 3 servers. Their optimization goal is to improve the overall system scalability and increase the throughput. They consider transaction support in their system

where all data they access must be handled atomically and kept under the same service (ACID properties). Compared to our system, they do not assume that the query templates are known beforehand and each query can be answered with a single node. Although, as a result of their experiment, they noticed that the data structures are mostly queried by few query templates, which could help to simplify their task. They do not talk about a possible router logic and automatic physical layout generation but they note that web service denormalization does not have any problem with consistency.

Ganymed [20] uses a novel-scheduling algorithm that separates update and read-only queries using multiple nodes. Ganymed routes 'updates' to a main server and 'selects' to read-only copies. Their system offers scalability without partitioning the data and it does not impose any restrictions on the incoming queries. It uses full replication and routes updates to the main replica to increase throughput, reduce response time, and increase the availability of the system. Their algorithm handles transactions and it keeps the replicas consistent, guaranteeing the ACID durability. It uses RSI-PC (Replicates Snapshot Isolation with Primary Copy) scheduling algorithm that separates reads and updates and hides the inconsistency from the client. Comparing to our assumption that we know the incoming query templates beforehand, Ganymed does not impose any restrictions on the queries submitted and has transaction support. It does not assume that the workload is known. It separates ready-only and update transactions and routes them to the set of duplicates. Updates, inserts, and deletes are routed to the main replica and reads are routed to any of the read-only copies. Their main optimization goal is the same as ours: to increase the throughput and reduce query response times. They realized that the communication cost between replicas can be really high and there can be deadlocks

involved at the scheduler level. Their scheduler takes care of inconsistency of the replicas and all synchronization is done transparently. They suggest an interesting solution to use multiple schedulers instead of a central one. This solution is different than ours but did not improve their performance. Read-only queries are assigned to a valid replica. A replica is valid if the latest writeset is committed. The scheduler keeps a connection pool open for each replica for serving the writesets.

[21] introduces an edge service architecture (edge refers to a component that intends to improve the performance of a Web-based system and distributes web content over the Internet) to improve the availability and performance of the Web-based applications by replication not just in a clustered environment but at geographically distributed sites. Their architecture is different than our system because we focus on the clusterized environment. Their system does not know about query templates beforehand but they try to answer each query locally using a single edge server. Their main optimization goal is to demonstrate that object-based data replication minimizes communication cost across the wide area network between database servers. Their main goal is to dramatically improve both availability and performance of the system with an object-based data replication. In object-based replication, data and business logic are replicated together on edge servers. They introduce different objects to handle one-to-many and many-to-one updates to propagate changes to multiple servers for keeping the consistency. They consider solving the update propagation problem across edge servers as a future work.

GlobeDB [22] offers a different approach for edge servers to handle data distribution. They replicate the data along with its access code across machines only if

the update rate is high enough at the specific location. The optimization goals are to figure out which part of the database needs to be replicated, find the appropriate place for the database part in the wide-area network, and keep the databases consistent. Their system automatically partitions and replicates the database through wide area network using multiple nodes. Their system is different than our middleware's infrastructure because we are operating within a cluster environment. GlobeDB realizes that replicating data on all servers can be a serious bottleneck. Their optimization goal is to place the data only on the servers that most frequently access them and increase the query response time. Their system focuses on the consistency issue as well with update propagation and concurrency control. They update the replicas immediately as soon as an update happens. GlobeDB follows a master-slave protocol where a master server is responsible for UDIs and propagating them to the replicas. GlobeDB does not assume that it knows the query templates beforehand. It tries to detect the complexity of a query (e.g. simple select, not an aggregate query) and use local replicas when it is possible to answer it. For complex queries, it forwards the query to a subset of servers that jointly have the complete database. GlobeDB does not assume that each query can be answered by a single node. They realized that the majority of web applications use simple queries which is a step toward our assumption. Some researchers make claims that their system will function well, if the real load is at least close to the load used for testing. Knowing the load ahead of time is not something that differentiates our work from others.

Several techniques are known for distributing the load across multiple database servers. One of these is replication [2]. In replication a table is placed on more than one database server. In such a case, a select query on the table can be executed by any one of

the database servers that have a replica of that table. An UDI query on that table however, needs to be executed on all the database servers that have a replica of that table. A drawback of this technique is that every UDI query needs to be executed against the node(s) that hold all of the entire data and these nodes become the bottleneck of the performance.

Our data replication middleware will detect which tables need to be replicated based on the given workload and the known query templates. The system will keep track of the reads and writes ratio for each data table and will determine the possible candidates for replication.

Master-slave architecture is supported by a couple of database systems [23][24] where there is a single master server that holds all of the data and every UDI query is executed against the master node and propagated to slave nodes as necessary. In this case, the master server is a real bottleneck of the system. Moreover, the synchronization cost of the slaves with the master database can be an issue as well. In a master-slave environment, all writes and updates must take place on the master server and reads can take place on one or more slave servers. This model can significantly increase the performance of reads. Its obvious that setting a single database node to be the master for all tables has a bottleneck, in that all UDI have to go to the same server. Large number of UDIs can cause a serious synchronization problem of the slaves, especially when the system has multiple slaves. There are two types of synchronizations: asynchronous and synchronous. Asynchronous data propagation happens immediately and it can take a relatively long time to write the data on all slaves and the client must wait for the propagation to happen. In the asynchronous scheme, data is written to the master but may

not be sent to the slaves until a certain elapsed time has passed. The downside of this schema is the possibility of data loss if the data has not been propagated to the slave when a critical fault occurs. In our system, if we shift the control over to a different table to different nodes we could possibly distribute the UDI bottleneck.

However, other master-slave architectures are possible where there is more than one master node [3]. DBFarm does not make the assumption we make that all queries are known ahead of time but they answer all read-only queries using a single replica and write-queries with one of the master databases. We are similar to [3] in that we will support full replication across serves, but we are different in that we will support horizontal and vertical partitioning, and denormalization. They simply separate reads and writes transactions where writes are performed at the master level and reads at the slave level. Read-only transactions executed at the slave databases are able to see all updates of the master database. DBFarm handles commit acknowledgements and assures read-only consistency. The drawback of this architecture is that writes have to happen on all masters. Write propagation can introduce a significant overhead and decrease the system throughput. Another technique for distributing load across multiple database servers in web applications is partitioning of data, which includes both horizontal and vertical partitioning. Horizontal partitioning splits the table up into multiple smaller tables containing the same number of columns, but fewer rows. Smaller partitions can speed up query performance if data needs to be accessed from only one of the partitions. Vertical partitioning splits the table into smaller ones with the same number of rows but fewer columns. It is a reasonable approach when the system does not want to combine the records between the partitions. To handle correct partitioning the system needs an

application logic to maintain partitions and balance the queries. Moreover, the horizontal and vertical partitioning problems over set of processors have been shown to be NP hard [4] and each operator faces a large search space. Paper [4] also describes a first-fit heuristic greedy algorithm to solve horizontal and vertical partitioning problems using multiple processors that are clusterized at one location but it does not assume that the workload and query templates are known. Moreover, it does not talk about any system or application logic that can maintain the created data structures. Some other researchers have done horizontal and vertical partitioning similar to our work, but in a different way. A Case for Fractured Mirrors [12] combines partitioning with multi-level cache and mirroring functions at the hardware level (e.g RAID1). This work is at the hardware level. Their work is similar to ours to the degree that they use vertical and horizontal partitions to try to get a speed, but its quite different in that it does not apply to multiple nodes, nor in the partition the same, even though it is called partitioning. This work is at the disk level, while ours is at the database level. While their work is interesting, its in use of vertical partition paying attention to which hard-drive cylinders are used to store the data but its similarity stops there.

Appendix B shows the comparison of the different systems.

2.4. Hybrid Solutions

MapReduce [50] is a programming model with an associated implementation to process and generate huge amounts of data in large scale (hundreds and thousands of nodes) heterogeneous shared-nothing environment. Shared-nothing environment deploys the

servers with their own local disk and memory connected with a high-speed network. MapReduce is a simple model consisting of only two functions: map and reduce. Users have to write these functions to produce key-value pairs based on the available input data. It uses a distributed file [51] system where the input data is partitioned and stored on multiple nodes of the cluster. The map function works like a filter or transformer operator that is applied on the input data set. The output of the map operator is a set of intermediate key-value pairs stored on the local disk of the node. These intermediate key-value pairs are partitioned into R disjoint buckets based on a hash function on the key of each output record. In the second phase R instances of the Reduce function are executed. The input files for Rs are transferred through the network from the local disk of the previous nodes where the Map function saved them. Reduce function processes and combines the input records and writes them back to the distributed file system in an output file. Parallel databases use shared-nothing infrastructures in a cluster environment and it can execute queries in parallel using multiple nodes. One big difference is that parallel databases support SQL (Structured Query Language) and standard relational database tables. MapReduce has no pre-defined schema and it allows the data to be in any format. Because the data can be in any format the system does not provide e.g. built in indexes.

However, HadoopDB [52] (Figure 2) combines the two approaches into one and targets the performance and scalability of a parallel database and the fault-tolerance feature of the flexible MapReduce to achieve better structured data processing. It uses Hadoop [53] the open source implementation of MapReduce to parallelize the queries across nodes. Scheduling and job tracking is managed by the Hadoop task coordinator

(JobTracker and TaskTracker). HadoopDB provides a front-end for users to process SQL queries. Queries are created using SQL-like query language (HiveQL [54]) and translated into MapReduce jobs by the help of the extended version of the Hive warehousing solution [54], called SMS planner. HadoopDB uses PostgreSQL [55] as a database layer that processes the translated SQL queries. To design a hybrid infrastructure like HadoopDB multiple key issues should be considered at different levels.

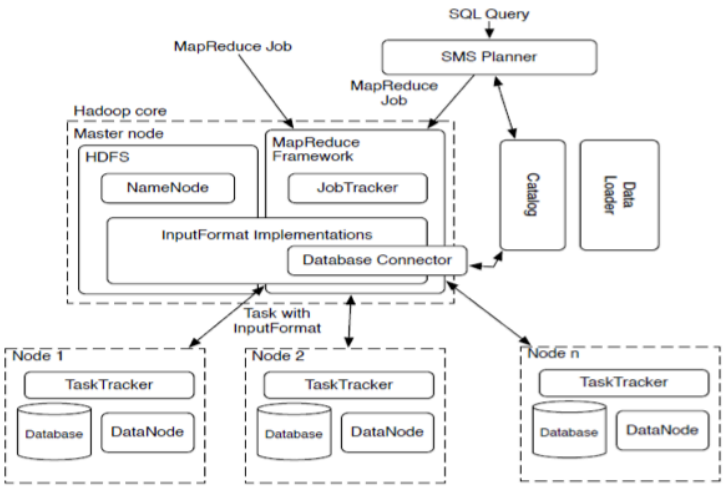


Figure 2: Architecture of HadoopDB [53]

To process extremely large datasets on a large-scale (thousand of nodes) shared-nothing environment where analytical workload performs heavy table scans scalability, performance, and availability are important factors. The amount of the analyzable data is growing and requires more and more computational nodes to complete the data analysis within a reasonable amount of time. Parallel databases can scale-up well if the number of the involved nodes is small. They assume a homogenous set of machines, but on a large

scale this assumption fails. In a heterogeneous environment the probability of a possible server failure is high. Google reported 1.2 failures/analysis job [50] for MapReduce. One of the first key issues is the data distribution. A highly scalable distributed file system is necessary to handle large amount of data and to eliminate a possible performance bottleneck. This file system should be fast and fault-tolerant for a possible node failure. HDFS (Hadoop Distributed File System) stores the data in fixed sized blocks and distributed across multiple nodes. A central service (NameNode) maintains information about the location and size of each chunk. The parallel query optimizer of a parallel database always sends the query to the node where the data is located. MapReduce always moves the data where the computation is performed. This hybrid infrastructure takes this data placement one step further. It loads the data from HDFS to the PostgreSQL nodes by the help of a dataloader that enforces two hashing phases. It utilizes the databases by dividing the data into as many chunks as the number of nodes. In the second phase it divides them further into chunks and loads each chunk into a separate database using a node. This structure is distributed thereafter. It can happen that some tables are collocated and partitioned on the query's attribute. In this case the requested operator can be handled by the database layer (PostgreSQL) directly. The implemented service (dataloader) should be fast and accurate to perform the two phase hashing, data re-partitioning, and loading the chunks. The SQL query interpreter and translator can be a performance bottleneck as well. This part of the system is responsible to optimize the query plans and translate SQL queries into MapReduce relational operators. The translator should be able to translate MapReduce relational operators back to SQL queries and utilize the parallel database at the database layer. Moreover, the SMS planner of the

system can be a huge single-point-of-failure. The SMS planner extends Hive and it is responsible for updating MapReduce's central information bank with the location of the database tables. It also scans the generated MapReduce jobs to determine the partitioning key in the DAG. SMS planner is a key component of the HadoopDB. If this component is not designed perfectly the system cannot interact with MapReduce. Fault tolerance is another key issue. If MapReduce is combined with parallel databases then it provides a more robust and sophisticated failure mechanism. If a job fails during execution the MapReduce framework can continue the failed job on the same node since output files of the Map function are kept on the local disk. If the node has a hardware failure then the system can re-schedule the task on a different node automatically. Parallel databases will not save intermediate results to disk and cannot continue the job execution. In the case of a possible node failure the database is capable to commit transactions successfully (e.g. log based commit).

As we have already mentioned above, parallel databases are not designed for usage in a large-scale heterogeneous environment. Concurrent queries, node disk fragmentations, or corrupted data parts can decrease the performance of the system. MapReduce can handle these problems as well. It can schedule parallel execution of the same task on different node if it detects that the data processing is slow. Furthermore it can catch the process before it terminates due to a bad data segment and re-schedule the task using data from a different location. These are all important designing issues. Further design issues are load sharing within the system and locality tracking. Load sharing should balance the load equally utilizing all the nodes. Locality keeps the slave processes close to the master process to reduce the communication overhead. MapReduce uses

master-slave topology to process the job. The master monitors the slave processes. Finally, this complex system should be flexible and allow users to write their own user defined functions that can be executed parallel to utilizing the databases.

There is a conceptual difference between parallel database management systems (DBMS), MapReduce-based systems (Hadoop), and hybrid systems like HadoopDB. In the case of DBMS, users can state what he/she wants in SQL language. In the case of MapReduce-like systems the user can present an algorithm to specify what he/she wants in a low level programming language [56]. HadoopDB hides the latest one from the users and provides the DBMS flexibility to the data analyst. In general there are a couple of differences between this MapReduce-based hybrid system and other parallel databases. DBMSs have pre-defined table schemas with rows and columns. MapReduce does not have any pre-defined schemas so the user has to create them. Once such schema is defined then the next task is to make sure no specific constraints are violated by the programmers. DBMS provides this check by default. DBMSs have the capability to create indexes (B-Tree/Hash-based) on specific columns to speed up scan functions. MapReduce does not have any built-in indexes. MapReduce introduces additional network traffic and disk accesses with the Reduce function that transfers and groups data parts together. Users can implement different functions in the Map and the Reduce parts, but DBMSs support user-defined functions, which can be executed in parallel. MapReduce HadoopDB combines all features of the DBMS with MapReduce to create a hybrid system that is good for analytical purposes.

Vertica Analytic Database [57] utilizes cheap shared-nothing commodity hardware and it is designed for large-scale data warehouses. It uses a column store architecture where each column is independently stored on different nodes. It applies vertical partitioning on the original dataset to create multiple partitions that can be replicated across cluster nodes (Figure 3). It is mostly for read intensive analytical applications where the system has to access a subset of columns. Vertica's optimizer is designed to operate on this column-partitioned architecture to reduce I/O costs dramatically. It employs various data compression techniques to minimize the space requirements of the columns. The optimizer stores views of the table data in projections. The projection can contain a subset of the columns of a table or multiple tables to support materializing joins. Projections are created automatically by Vertica to support ad-hoc queries. To avoid node failure, Vertica creates $k+1$ copies of the projections (k is the total number of nodes) and fully replicates them. In the case of a failure, it automatically switches to the next available instance. It has a built-in automatic physical database design tool that creates these projections automatically and targets star (fact and dimension tables where fact tables are range partitioned across the nodes and dimension tables are replicated) or snowflake (normalization of dimension tables) schemas for automatic design.

The hybrid storage module caches all updates to a memory segment called WSO (Write- optimized Store). A tuple mover migrates recent updates to permanent storages periodically and the system uses snapshot isolation to keep the consistency with the current updates. HadoopDB is a more robust system that provides fully structured relational tables with SQL language support where the structured data can be optimized.

Vertica has a built-in automatic database design tool that can adjust the system performance creating a new projection or partitioning the data according to the preferable schema.

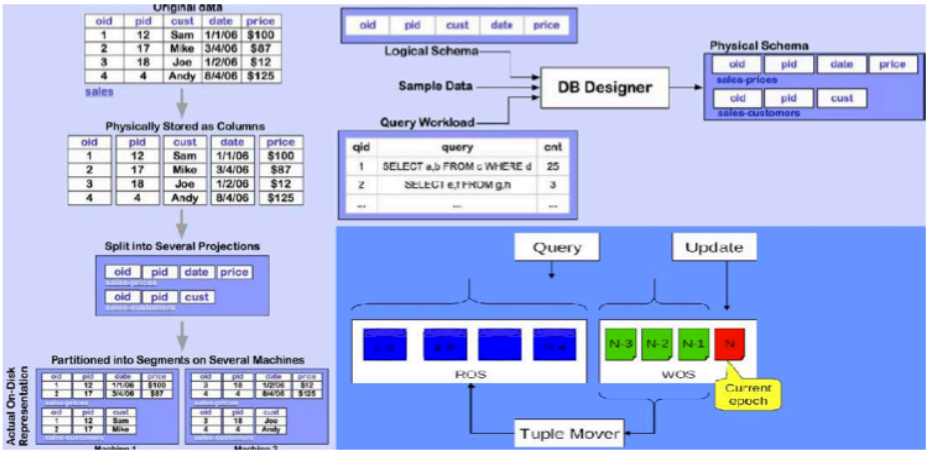


Figure 3: Architecture of Vertica [57]

Vertica has nice data compression techniques that outperform HadoopDB in an I/O intensive task. HadoopDB SMS planner pushes the different SQL clauses into the database layer and it can benefit from the created table indexes. A problem can be that the data is not partitioned according to the requested key. In this case the hybrid infrastructure re-partitions the data that can decrease the system performance significantly. In the case of a join task – that execution time can be crucial – the two systems have a big difference. Vertica with the data compression, indexing, and projection has a native built-in join query support. HadoopDB SMS optimizer (based on Hive) does not have full support for joins and cost-based optimization of the queries. HadoopDB can benefit from the join if both input datasets are sorted on the join key. Then it can push down the join to the database layer. Otherwise, re-partitioning of the

input data is required that adds significant overhead.

Netezza [58] parallel system is a two-tiered system that is capable of handling large queries from multiple users. The first-tier is a high-performance multiprocessing unit that compiles queries and generates the query execution plans. It divides the queries into sub-tasks for parallel processing and utilizes the second tier's Snippet Processing Units (SPU) for execution. Netezza combines the two tiers into one and hides the complexity of the system while providing SQL interface to the users. In the case of HadoopDB indexing can speed-up the queries execution time. Netezza has no indexing feature because the query processing is done at the disk level. The proper distribution of the tables over SPUs is the key issue to achieve high performance. The system distributes the tables based on the fields that would be indexed by HadoopDB and each SPU can process its own set of data without intercommunication with other SPUs. Teradata [59] has two main requirements: ensure that the data is available when it is requested and being able to access the information without significant delay. It uses a shared-nothing architecture where the data is assigned to each unit. Virtual Access Module Processes (VAMPs) is responsible for controlling the database processing. VAMP executes index scans, reads, join, etc. functions using its own independent file system. A difference is that Teradata supports single row manipulation, block manipulation and full table or sub-table manipulation as well. It distributes the data randomly utilizing all the nodes. It provides a single hash-based partitioning algorithm that partitions the data equally across all VAMPs. The hash re-distribution is an automatic task in the background according to the required update, delete, or insert actions. Another big difference is that it has a built-in dynamic statistics collector that dynamically increases the number of VAMPs upon

high load and distributes the requests equally. It has a built in optimizer that can handle sophisticated queries, ad-hoc queries, and complex joins as well. Teradata supports direct data loading into the database and the system handles the partitioning, indexing, etc. automatically.

IBM DB2 Data partitioning [25] is based on shared-nothing architecture as well. Their tool suggests possible partitions based on the given workload and the frequency of each SQL statement occurrence. Their system recommends the best operator (partitioning or full replication) for each table. The system computes a set of interesting candidates that can help to reduce the query cost (overall cost is a combination of different constraints: I/O, CPU, and communication cost). There are several other parallel databases available like Exadata (parallel database version of Oracle), MonetDB, ParAccel, InfoBright, Greenplum, NeoView, Dataupia, DATAllegro, Exasol, etc. that all combine different techniques to achieve better performance and reliability.

Our parallel database architecture [42] is also based on shared-nothing community hardware where each node has its own CPU, disk, RAM, and file system. We specialized on Web-based application where the workload consists of a fixed number of query templates. This means the system does not face ad-hoc and unexpected queries. Because we know all the query templates beforehand our system can pre-partition the data using different operators and pre-determined heuristics [60]. Each node has a relational database but the main difference is that we do not need to re-partition the data like HadoopDB since we do not have unexpected queries. We characterize the problem as an AI search over database layout. We iteratively minimize the total cost of the workload creating different database layout and increase the system throughput (model and

partition first then load the data approach vs. load the data and re-partition). Moreover, our system has a built-in corpus with generalized machine learned rules to determine which operator (Horizontal partitioning, Vertical Partitioning, Replication, and Denormalization) is applicable and when. As soon as the layout is determined, the data is distributed across the server nodes. A central dispatcher – similar to HadoopDN catalog - maintains the statistics about the current layout (table descriptors, data part locations, etc.). Since we do not have an unexpected query, the data can be partitioned according to a pre-defined rule: each query should be answerable using a single node. This means that all the joins can be pre-computed and the communication bottleneck (e.g. in the case of MapReduce the Reduce function moves the files and loads the data from multiple location) can be eliminated. The central dispatcher can push each query into the database layer directly where the well-defined schemas support indexing. We support INSERT INTO, UPDATE, and DELETE SQL statements natively (Hadoop with Hive does). We do not have an additional failure detection mechanism, but the system is easily expandable with a full copy of the original database. Furthermore, our system needs an additional layer – possible integration with Hadoop - if it wants to scale-up to thousands of nodes.

The next chapter will talk about the data placement problem, problem statement, optimization goal, data placement algorithm, and the state space search.

3. Data Placement

Scaling up web applications requires distribution of load across multiple application servers and across multiple database servers. Distributing load across multiple application servers is fairly straightforward; however distributing load (select and UDI queries) across multiple database servers is more complex because of the synchronization requirements for multiple copies of the data. Different techniques have been investigated for data placement across multiple database servers, such as replication, partitioning and denormalization. In this chapter, we describe our framework that utilizes these data placement techniques for determining the best possible layout of data. Our solution is general, and other data placement techniques can be integrated within our system. Once the data is laid out on the different database servers, our efficient query router routes the queries to the appropriate database server/(s). Our query router maintains multiple connections for a database server so that many queries are executed simultaneously on a database server, thus increasing the utilization of each database server. We have implemented our solutions in our framework.

There are thousands of web applications, and these systems need to figure out how to scale up their performance. Web applications typically have a 3-tier architectures consisting of clients, application, and a database server that work together (Figure 4). Significant work has been done in load balancers to solve possible scalability issues and to distribute requests equally among multiple application servers. However, issues related to the increased database server usage and to distribute requests among multiple database servers have not been adequately addressed. The increasing load of the database layer can

lead to slow response time, application error, and in the worst case, to different types of system crashes.

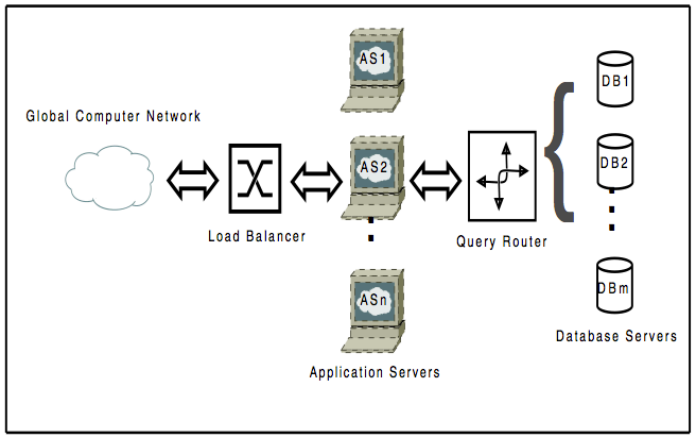


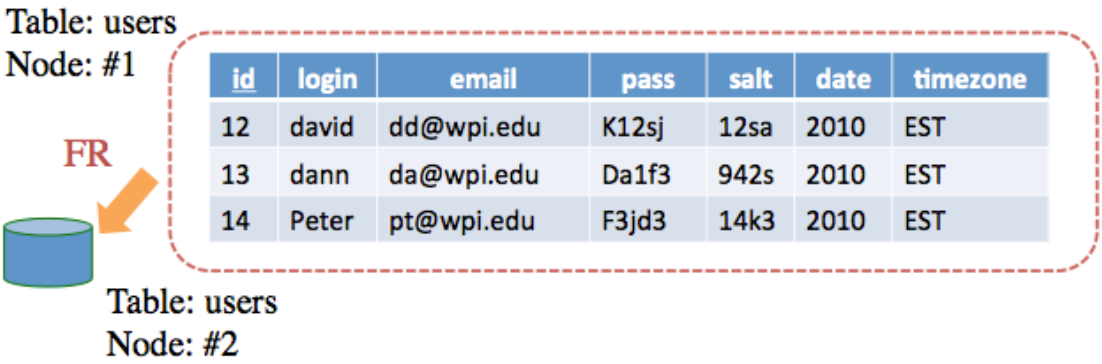
Figure 4: General Architecture of a Web-based application
The requests are distributed among the different application servers by the load balancer. Requests that need to access the data are sent to the query router, that routes the query to the appropriate database server(s).

In a Web-based application the increasing number of user sessions can be easily balanced among application servers but the continuous database read (select) queries, and update, delete and insert (UDI) queries decrease the system response time significantly.

3.1. Current Techniques for Distributing Load

Several techniques are known for distributing load across multiple database servers; one of them is replication [2]. In replication, a table is placed on more than one database server (see Figure 5). In such a case, a select query on that table can be executed by any one of the database servers that have a replica of that table. An UDI query on that table however needs to be executed on all the database servers that have a replica of that table.

If we do not know all the queries that the application may need to process beforehand, then one of the database servers must hold the entire data (all the tables) of that application. Such a layout of the data is needed to answer a query that needs to access all the tables. A drawback of this technique is that every UDI query needs to be executed against the node/(s) that hold the entire data and thus these nodes can become the bottleneck for performance. Such an architecture is supported by Oracle, and is referred to as a master-slave architecture. In this case, the master node holds the entire data; every UDI query is executed against the master node and propagated to slave nodes as necessary using log files.



id	login	email	pass	salt	date	timezone
12	david	dd@wpi.edu	K12sj	12sa	2010	EST
13	dann	da@wpi.edu	Da1f3	942s	2010	EST
14	Peter	pt@wpi.edu	F3jd3	14k3	2010	EST

Figure 5: Full replication of a table

In the case of web applications, we no longer need a node that holds the entire data (assuming that none of the queries access all the data). We can therefore do a more intelligent placement of the data such that there is no node that must execute all UDI queries; thus we can remove the bottleneck node for UDI queries that is inherent in non-web applications. This placement improves performance of read queries while not significantly impacting the performance of UDI queries. In case of full-replication (all nodes are effectively master nodes), any node can act as a master when the original master fails, and the routing of queries to the nodes is straightforward as any node can

answer any query, but the updates have to be propagated to all of the nodes.

Another technique for distributing load across multiple database servers in web applications is partitioning of data, which includes both horizontal and vertical partitioning. Horizontal partitioning (Figure 6) splits the table up into multiple smaller tables containing the same number of columns, but fewer rows.

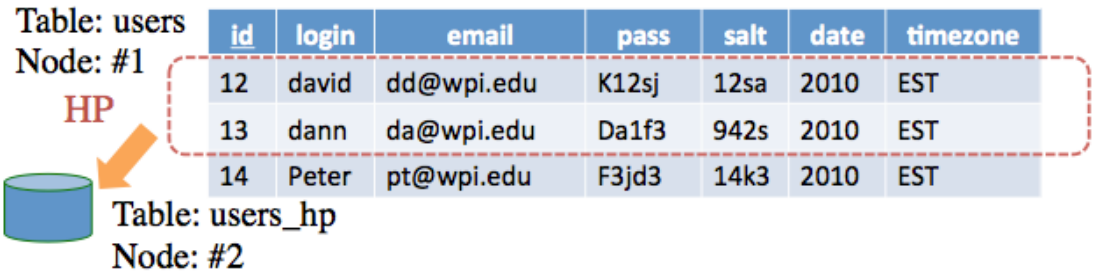


Figure 6: Horizontal Partitioning of a table

This technique can speed up query performance if data needs to be accessed from only one of the partitions. However, horizontal partitioning cannot be done in all circumstances, if we want a query to be answered by one of the nodes. For instance, if there are two queries in the workload that access the same table, one which selects based on a column say C1, and another which selects based on a column C2, then if we do horizontal partitioning based on the values in C1, then this partitioning cannot be used to answer queries based on C2. Vertical partitioning (Figure 7) splits the table into smaller ones with the same number of rows but fewer columns. It is a reasonable approach when the system does not want to combine the records between the partitions. Another big job for both the partitioning schemes is that the system needs to maintain the partitions and balance the amount of data with a built in application logic.

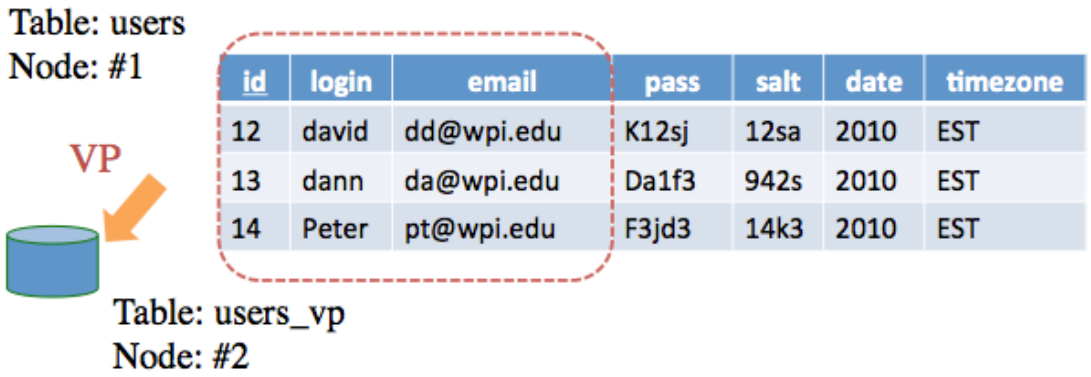


Figure 7: Vertical Partitioning of a table

Denormalization [40] can optimize the performance of database systems as well. In denormalization, one moves from higher to lower normal forms in the database modeling and add redundant data (Figure 8). The performance improvement is achieved because some joins are already pre-computed. However, there is more complexity involved. For instance, handling UDI queries are more complicated when performed against denormalized data, as we need to synchronize between duplicates. Also, the routing logic needs to be more advanced.

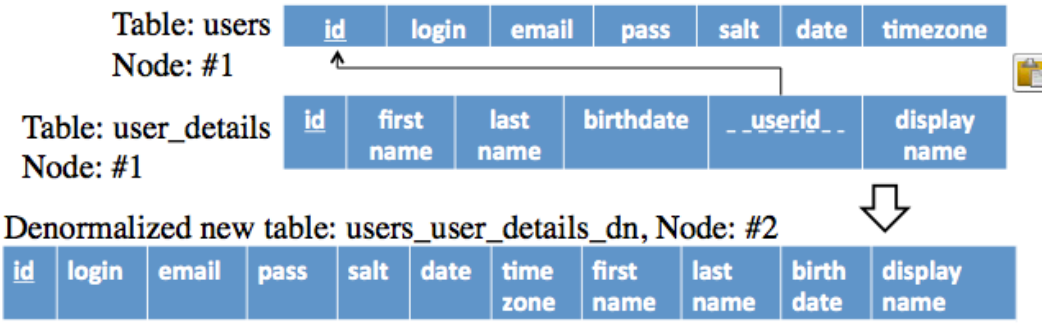


Figure 8: Denormalization of two tables

3.2. Data Placement Problem

Let us now define the data placement problem as: we are given all the query templates for a Web-based application, the database constraints, and a query workload (W) that includes the percentage of queries of each template that the application typically processes. Determine the best possible placement of the tables on the different database server nodes minimizing the total system response time (T) [60].

We conceptualize the database layout problem as a state search problem where each state is a valid configuration of layouts across database nodes.

Let L = {L1..L_n} be the set of possible valid layout configurations of the tables on the different server nodes. Let cost(Q, L) be the total system response time with query workload (W) using a valid layout configuration. Find the best valid configuration such that:

$$MIN \sum_{i=0}^{n} cost(Q_i, L) \qquad \textbf{(equation 1)}$$

Definition 1: Valid state. A state S_n is considered to be valid if and only if the created layout configuration L_i correctly answers each and every query Q_i from the given workload W.

A query template maps a query to one or more SQL statement. For example 'SELECT * from problem_logs where problem_logs.id = 12' and 'SELECT * from problem_logs' are two different query templates of the same table. A query workload is any group of queries that run on the database (percentage of queries for each query template).

We leverage the fact that the workload of a Web-based application contains only a small set of read and update/delete/insert (UDI) query templates (typically between 10 and 100) [1] [14] [17]. One detailed architecture for a Web-based application is shown in Figure 9. First, the data is placed on different database servers. Different clients connect and issue requests, which are distributed across different application servers by the load balancer. Balancing the load across different application servers can be done effectively by scheduling the requests using simple schemes such as round-robin, or scheduling the next request on the current least loaded server; these are not discussed further in this paper. A request may need to access data in the database server, in which case a query is issued to the query router. The query router has the logic to route the queries to the appropriate database server/(s). In short, the query router maintains the information about how the data is placed across different database servers.

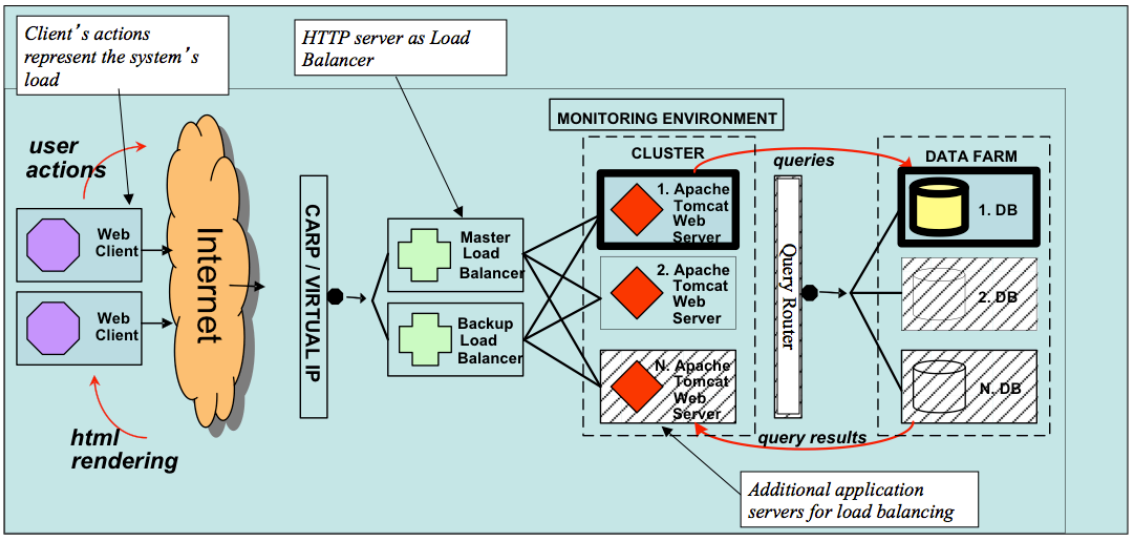


Figure 9: Detailed architecture of a Web-based application

Let us motivate the data placement problem using a thinned down schema. The portion of

the schema that we consider includes users (students), schools, user roles (that maintains the school that a user attends), problems and logged action (that maintains all the actions of every user, including logins of a user, problems that a user has attempted). Let's define 16 query templates for illustration as shown in Table 1. Note that for illustration purposes, we used only simple queries that do not perform a join. These data were collected over the duration of one week from a real Web-based application [62], and we counted the number of queries for each template. The total number of queries for these 16 templates over the week was about 360,000. We also have shown the number of rows of each table, at the end of the week over which the data was collected. Before we describe our data placement algorithm, let us examine Table 1 closely, and study what issues the placement algorithm may have to tackle.

Table 1: Example Illustrating Query Templates and Workload

# of rows denotes the number of rows in the table accessed by the query.

Query	Template	Table name	% of queries	# of rows
1	SELECT * FROM schools WHERE school.id=?	schools	<1%	321
2	SELECT * FROM schools WHERE schools.name=?	schools	<1%	
3	SELECT * FROM schools	schools	<1%	
4	SELECT * FROM users WHERE users.id=?	users	19%	30826
5	SELECT * FROM users WHERE users.login=?	users	<1%	
6	UPDATE users WHERE users.id=?	users	<1%	
7	INSERT INTO users	users	<1%	
8	SELECT * FROM problems WHERE problem.assignment id=?	problems	13%	20566
9	SELECT * FROM problems WHERE problems.id=?	problems	15%	
10	SELECT * FROM problems WHERE problems.scaffold id=?	problems	<1%	
11	UPDATE problems WHERE problems.id=?	problems	1%	
12	DELETE problems WHERE problems.id=?	problems	1%	
13	SELECT * FROM user_roles WHERE user roles.id=?	user_roles	19%	42248
14	INSERT INTO user_roles	user_roles	<1%	

15	UPDATE logged action WHERE logged_action.user id=?	logged_action	16%	7274174
16	INSERT INTO logged_action	logged_action	16%	

As there are many updates against the logged_action table, if logged_action is replicated, the costs of performing these updates will high. Instead it might be better to perform a horizontal partitioning of the logged_action table and place the different partitions on the different database server nodes. We notice that there are lots of updates against the problems table as well (ratio of UDI queries to select queries is roughly 1:14). However, Q8, Q9 and Q10 all access the problems table, but perform selects on different columns (Q11 and Q12 use the same column as Q9). In this case, we may want to consider maintaining only one copy of the problems table (rather than replicating the table or horizontally partitioning the table). Once a table is placed on only some of the database server nodes, the workload on the different database servers may now be high. For instance, suppose problems table is placed on node 1, there is additional load on node 1 as compared to the other nodes. This placement may impact the horizontal partitioning.

3.3. Data Placement Solution

In this chapter, we describe our algorithm [60] that given any query workload determines the best possible placement of the tables. Our data placement algorithm (DPA) is shown in Figure 10.

Step0. Apply Initial Data Distribution Policy: fully replicate all tables across all nodes

Step1. Determine the cost of the workload by running each query template on lightly loaded nodes.

Step2. Initialize an array **dataLayout** that maintains the current data placed on each database server.

Step3. Initialize an array, **currCost** that maintains the current system response time.

The initial cost is set to the result of Step1.

Step4. For each <query template, table> pair, initialize **setOfOptions** to all possible options.

// for instance **setOfOptions** = {replication, horizontal partition, vertical partition, de-normalization}

Step5. For every query in template, remove invalid options from the **setOfOptions**.

Step6. Iterate through the list of query templates and for each query template,

Step 6.1. "Search" for the best possible placement for every table in the query.

Step 6.2. Update the **dataLayout** array to indicate the data on each database server after this placement.

Step 6.3. Update the **currCost** array to indicate the current system response time after this placement.

Step7. Layout the tables across the different database servers according to the **dataLayout** array.

Figure 10: Data Placement Algorithm (DPA)
The dataLayout array returns the best possible layout of the tables across the different database servers.

Let us examine this data placement algorithm in detail. The **dataLayout** is the data structure that returns the best possible placement as determined by our algorithm. First, a <query template, table> pair (described in Step 4) consists of the table that is accessed by the template. For instance, for Q1 in Table 1, we consider <Q1, schools>, whereas for Q4, we consider <Q4, users>. For a join query, say Qi that joins tables T1, T2, we consider <Qi, T1> and <Qi, T2>. Also, the set of options described in Steps 4 and 5 can be modified based on what options are suitable for a specific application.

3.4. State Space Search over Layouts

We consider the database layout problem as a state space search problem with the assumption that all incoming queries should be answered by a single node. We do a time intensive search over different layouts, and each time, physically create the

configurations, and evaluate the total response time of the system. A state is a given assignment of tables to computer servers. The operators in the search are to fully replicate, horizontally partition, vertically partition, and denormalize a table. After each valid state creation the system measures the total response time of the system using the query workload to get actual performance measures from a real setup. Through our experiments and construction we used these real numbers in place of a possible estimator to demonstrate our algorithm's functionality. The search over layouts can be very expensive and a possible virtual partitioning (Chapter 6.4) or a DBMS optimizer (like IBM DB2) can be used to predict numbers as a replacement black-box component for actual performance measures. However, entities trying to scale up their Web-based applications would be perfectly happy to prefer real run-time measurements over estimated ones and spend a few weeks of CPU time to increase their system throughput. We determine the possible states based on the query templates and we do not consider states that are possible to further create but not be used in these templates. Figure 11 shows an initiated complete search. As a start state (state 0) we fully replicate all tables across all database nodes and measure the total response time of the system using the given workload. The system provides two algorithms to traverse the search tree: the naive and a simplified one-level search algorithm. The default search algorithm is the naive. We traverse down an entire path (state 1, 2, 3, and 4) before backtracking to the next valid path (state 2, 5, and 6). As soon as a valid state is created, the system measures the total system response time using the query workload. As one of the guiding rules we backtrack to the next valid path if the throughput of a child is less than the throughput of its parent. For example, if the throughput of state 18 is less than the throughput of state

17 then we will not explore states 19, 20, 21, and 22.

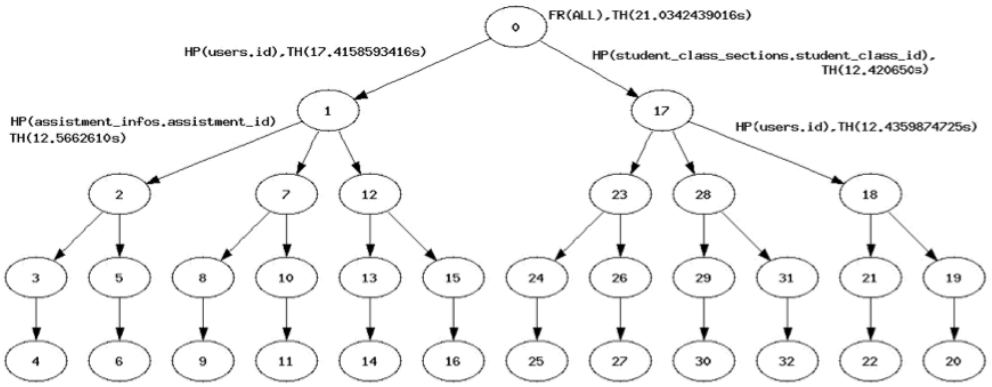


Figure 11: State Space Search

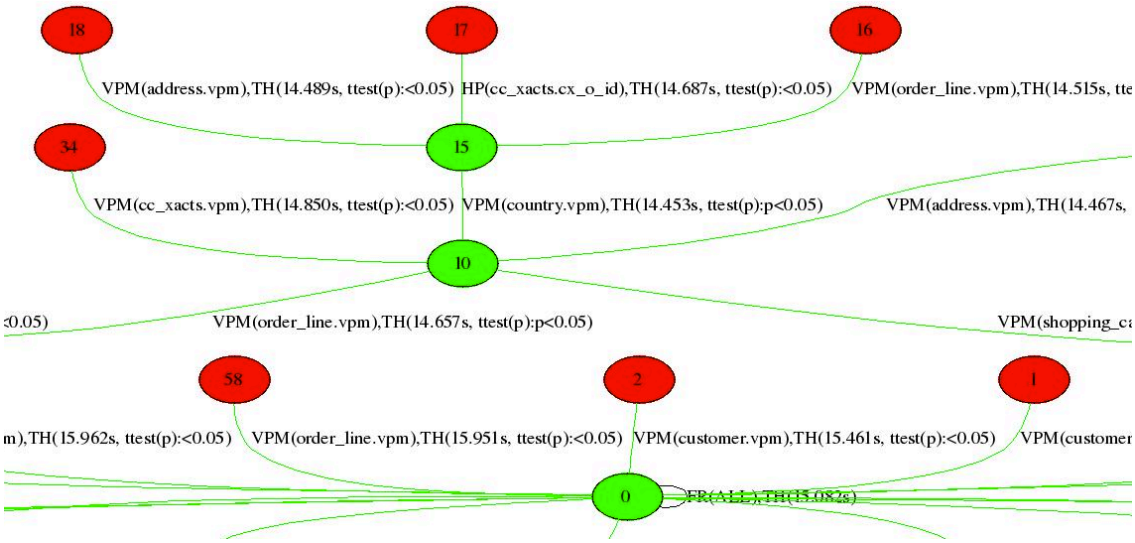


Figure 12: Result of an initiated layout search

Figure 12 shows a result of an initiated layout search. Green state means that the total system response time is significantly better (t-test) compared to the previous state's time. Red state means that the total system response time is significantly worse or not significantly better (t-test) than the result of the previous state. In this case we will not continue the search along that path and the algorithm backtracks.

The framework automatically executes the workload and measures the total system response time two times in default. Based on the measurements, it generates the p value of the results and marks the state green or red. The significance level is 0.05. The naive approach does not avoid us considering the same state over and over again. The one-level search algorithm simplifies the search by working at one level deep only. State 0 is the start state. At the next level, it considers all possible valid states one by one (see Figure 13). If one state is evaluated (eg. state 1) it does not continue along the same path. It backtracks to state 0 and removes the previously evaluated state (state 1) from its list. As a continuation it considers a never tried new state (state 2). The One-level search method eliminates the redundancy problem.

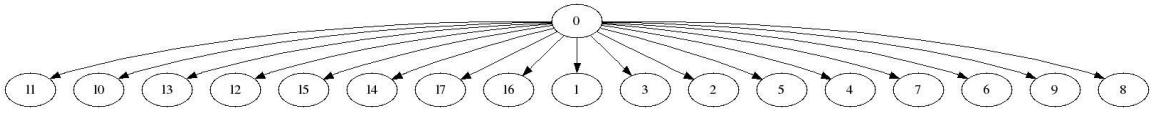


Figure 13: Simplified one-level search

3.5. Horizontal Partitioning

Horizontal partitioning is a logical database design technique that reduces the size of the irrelevant tuples accessed. This technique is used for distributing load across multiple database servers in web applications. Horizontal partitioning splits the table up into multiple smaller tables containing the same number of columns, but fewer rows. These splits can speed up query performance if data needs to be accessed from only one of the partitions. However, horizontal partitioning cannot be done in all circumstances, if we want a query to be answered by one of the nodes. For instance, if there are two queries in

the workload that access the same table, one which selects based on a column say C1, and another which selects based on a column C2, then if we do horizontal partitioning based on the values in C1, then this partitioning cannot be used to answer queries based on C2.

There are two important facts of the horizontal partitioning that we support, namely, partition localization for the given workload to generate a correct result set and answering queries using a single node. The problem of partitioning in the relational database systems has been recognized for its impact on the performance of the system as a whole [36], where one of the main optimization parameters is the number of accesses by the application to different parts of the data [37]. We addresses the horizontal partitioning problem for Web-based systems where all the incoming query templates are known beforehand and the retrieval queries should be answered by a single node. We propose a primary algorithm that searches for possible partitioning keys based on the query access patterns and based on the algorithm's optimized search space. Our horizontal partitioning algorithm supports simple predicates only. We introduce a definition before describing the algorithm.

Definition 2: **Simple Predicate** [36]. A simple predicate is a predicate defined on a simple attribute or a method and it is defined as: *attribute-method operator* value, where the operator is a comparison operator (<, >, <=, >=, <>, =). The value is from the domain of the attribute. The predicate evaluates to a logical value true, false, or unknown. For example, grade='A' returns true if the grade column contains the string A, false if the column has no 'A' value, or unknown if the grade column contains null. A predicate also can be range comparison (BETWEEN), inclusion test (IN), pattern match (LIKE), NULL test (IS NULL), and unique predicate (DISTINCT) but we do not

support these predicates for horizontal partitioning.

3.5.1. Operator and Framework Limitations

In the case of horizontal partitioning, our algorithm does not support range comparison, inclusion test, pattern match predicates, and <, >, <=, >=, <> comparisons on the partitioning key.

A typical SQL query can be represented as:

"SELECT [DISTINCT | ALL] column_expression1, column_expression2, ... [FROM from_clause] [WHERE where_expression] [GROUP BY expression1, expression2, ...] [HAVING having_expression] [ORDER BY order_column_expr1, order_column_expr2, ...]*"*

The "**WHERE"** clause can specify the limitations of the horizontal partitioning. The "**WHERE"** clause is not necessary if you want to retrieve parameters of all rows in a specified table, but this leads to not be able to consider horizontal partitioning operator without a "**WHERE**" clause.

The "**GROUP BY"** cannot be considered for horizontal partitioning because it terminates our assumption and we cannot answer the query using a single node.

"JOINS": We can join any two (or more) tables in the databases as long as they have some quantity or an appropriate relationship in common (e.g an ID). In this case, we have to check each joining condition and their relationship (see Chapter 3.5.3):

- We *must* specify all the tables on which we place constraints (including the join) in the FROM clause, but we can use any subset of these tables in the SELECT. If we use more than two tables, they do not all need to be joined on the same quantity
- The parser component of our framework does not have support for queries in the format of an implicit JOIN: "*SELECT * from user, progresses WHERE user.id=progresses.id*" and for three/multiple way joins in the format of "*SELECT user.ID, progresses.ID, teacher.ID FROM user u, progresses p, teacher t WHERE u.id = p.id and p.id = t.id*"

Aggregate functions and Mathematical operators: MAX, MIN, SQRT, POWER, AVG, EXP, LAST, FIRST, COUNT(*) without WHERE condition e.g. "*SELECT min(score),max(score) FROM user u group by score*" are not handled by our framework. We cannot handle Aggregate functions and Mathematical function without using more than one node to answer the query unless we are individually querying all database nodes and creating the union of the results.

OR operator: We do not support "OR" operators for horizontal partitioning since the "OR" keyword can request keys that was not used for horizontally partitioning a table.

Views: Systems with views (virtual subset of a table) are handled as normal table by our algorithm.

Nested queries: We do not support nested queries.

Transactions: We do not support transactions.

Union: We do not support the UNION operator.

We support hash-based partitioning only where the partition number for a given row is generated by a system specific hash function and it is applied on an object or objects of the table. This type of partitioning is defined by (O, H, n), where O is the involved objects or columns of a table, H is the system specific hash function, and n is the number of available nodes for partitioning.

For example, if table A has three columns (C_1 int, C_2 int, C_3 int) then the partitioning function defined by (A:C_1(int), H, 3) partitions table A into 3 partitions applying H hash function on the values of column C_1 in each row of table A.

If any queries do not meet these limitations then the horizontal partitioning operator is not viable as a choice for our algorithm.

3.5.2. Database Constraints

Database constraints define different rules regarding the values allowed in the database table or in the specific column. There are multiple types of constraints. A constraint can be defined when a table is created and modified later on. Typically there are five types of database constraints: primary key, foreign key, check, NOT NULL, and UNIQUE constraint. For us the most important ones are the primary and foreign key constraints because we can identify the relationships between two tables. Primary key constraint ensures that a column value is unique among all rows in a table and does not allow null values. Foreign Key restricts the values that are accepted in a column or columns and it

establishes a link between the data in two tables. It is the primary element needed to detect and analyze how to partition the tables. A foreign key points to the primary key of another table and the purpose of the foreign key is to ensure referential integrity of the data. A foreign key is a column or group of columns in one table whose values are defined by the primary key in another table [61].

For example, we have two tables: user and user_details

- All user details must be associated with an user that is already in the user table;
- We place a foreign key on the user_details table and have it related to the primary key of the user table;
- The user_details table cannot contain information that is not in the user table;
- The user table can contain information that is not in the user_details table;

If we have a join where user.user_id is a foreign key and it points to the primary key of user_details table user_details.id:

"*SELECT user.Lastname FROM user INNER JOIN user_details ON user.user_id = user_details.id WHERE user.user_id=1299*" //Retrievable if HP key is user.user_id

Only the columns of the user table can be considered as a key for horizontal partitioning.

"*SELECT user.Lastname FROM user INNER JOIN user_details ON user.user_id = user_details.id WHERE user_details.id=2349*" //Retrievable because the user_details table cannot contain information that is not in the user table and if the HP key is user.user_id

A foreign key relationship can be explicit or implicit. The explicit key relationships are defined in the database itself but the implicit ones (virtual) are not. Check constraint is a table level one and it restricts a column value to a set of values defined by the constraint. NOT NULL restricts a column and we cannot insert a row in the table without providing a valid data for the column. UNIQUE one will make the column value unique among all rows in the table.

3.5.3. Table Relationships

Foreign keys identify a relationship between two tables. Table relationships can be one-to-one, one-to-many, and many-to-many.

3.5.3.1. One-to-One

In a one-to-one relationship there is a single value in both directions. Each row in table A is linked to one and only one other row in table B. The number of rows in Table A must equal the number of rows in Table B (see Figure 14).

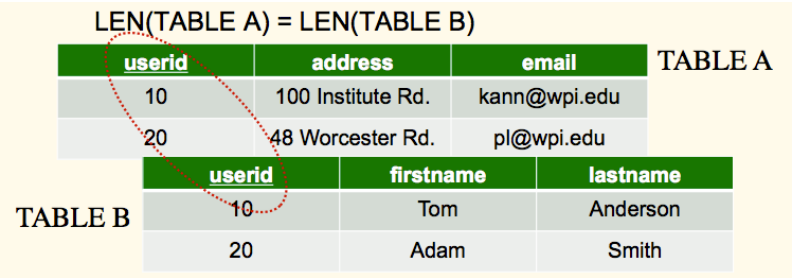


Figure 14: One-to-one relationship

Each row in table A is related to 1 and only 1 other row in table B and vice-versa.

3.5.3.2. One-to-Many

In a one-to-many relationship between Table A and Table B the rows in Table A are linked to zero, one, or many rows in Table B. This relationship allows information to be saved in a table and referenced many times in other tables. The total number of rows in Table B is almost always greater than the number of rows in Table A (see Figure 15).

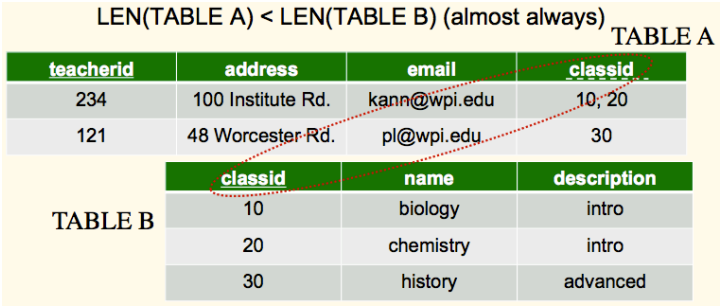


Figure 15: One-to-many relationship

Each row in the related table can be related to many rows in the relating table.

If we turn the relationship around then the relationship will be many-to-one.

3.5.3.3. Many-to-Many

In the case of a many-to-many relationship, each row in Table A is linked to zero, one or many rows in Table B and vice versa. Normally, Table C a mapping table is required to map such kind of relationships (see Figure 16).

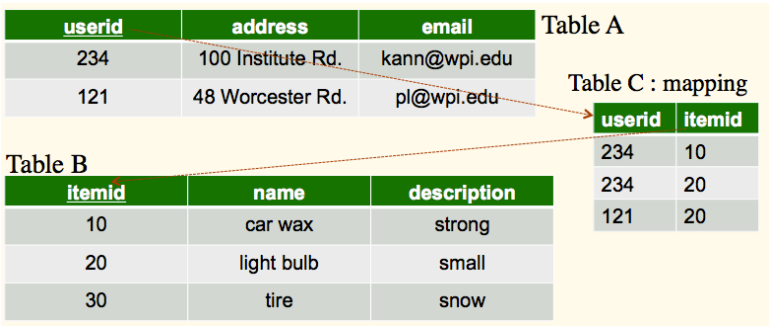


Figure 16: Many-to-many relationship

Each row in the related table can be related to many rows in the relating table and vice versa.

3.5.4. Partitioning Rules

3.5.4.1. Partitioning Table "A"

Definition 3: Horizontally partition table "A". We could horizontally partition table "A" if there is a key "X" and for "X" both of the conditions hold:

1. All the queries that join with table "A" use key "X" or a child key "Y" (child key "Y" has a one-to-one relationship or a many-to-one relationship with parent key "X") in the join;
2. All of the queries on table "A" have a "WHERE" clause that contains key "X" or a key "Y" that is a child of "X".

If there are more than one keys possible e.g. "X" or "Y", then we consider both keys for horizontal partitioning in a random order.

As an example for partitioning tables individually, we can mention when dimension tables of a star schema are large, they can each be normalized to create

multiple tables having a typical relational database design. The resulting variation of the star schema is called a "snowflake schema" shown in Figure 17. In the figure, the dimensions tables are decomposed into a snowflake structure to avoid joins to a large table. In some cases decomposed structure may improve performance because smaller tables are joined. We assume that queries in the workload can contain any subset of these foreign keys to primary key joins. We can horizontally partition the "Customer", "Parts" and "Supplier" tables individually based on X(1)="Custkey", X(2)="Partkey", and X(3)="Suppkey".

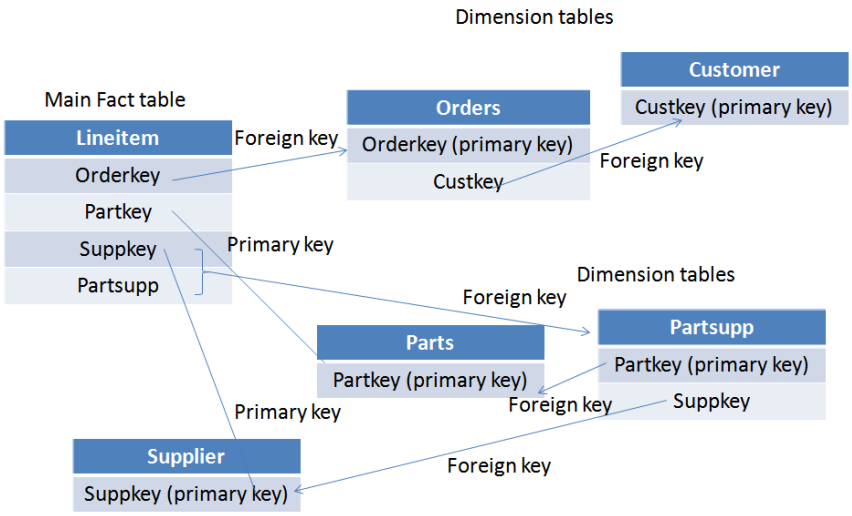


Figure 17: Snowflake schema

3.5.4.2. Partitioning Table "A" and "B" Together

Definition 4: Horizontally partition table "A" and "B". We could horizontally partition table "A" and "B" together if:

- Table "A" is nested in table "B" and;
- There is a key "X" and for "X" two conditions are hold:

1. All the queries that join with either table (“A” and “B”) use nested key “X” or a child key “Y” (child key “Y” has one-to-one relationship or a many-to-one relationship with nested parent key “X”) in the join;
2. All the queries on each table (“A” and “B”) have a “WHERE” clause that contains nested key “X” or a key “Y” that is a child of “X”.

Table A is nested with respect to table B if there is a one-to-one or a many-to-one relationship from table B to table A. Alternatively, table A is nested with respect to table B if:

- Table B has a column that is a foreign key to table A’s primary key, or
- Table B has a column that is a duplicate of one of the table A’s columns

Figure 18 shows an example for partitioning group of tables together.

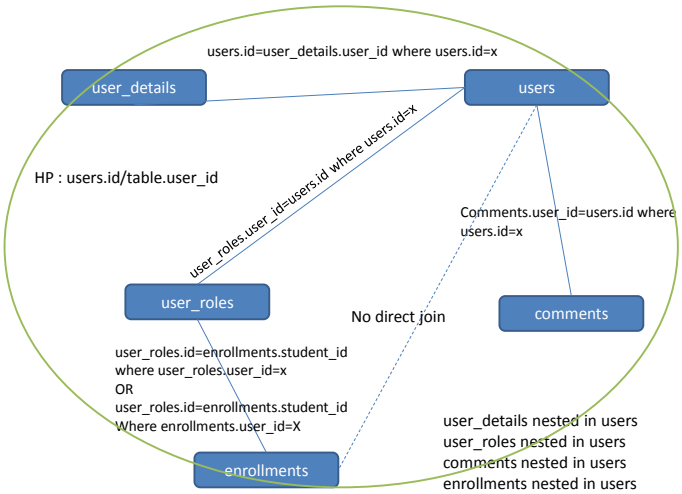


Figure 18: Partitioning group of tables

Appendix D shows the possible relationships between tables.

3.5.5. Partially Ordered Set

P = (X, P) is a partially ordered set – poset, if X is a set and P is a reflexive, antisymmetric, and transitive relation. X is the ground set and P is a partial order. We use the notation $x \leq y$ for $(x,y) \in P$ and $x < y$ if $x \leq y$ but $x \neq y$. We also use $x \rightarrow y$ for $x < y$. Two elements $x,y \in X$ are either comparable when either $x \leq y$ or $y \geq x$, or they are incomparable. A poset is a chain (or a totally ordered set or a linearly ordered set) if each pair of elements is comparable, and it is an anti-chain if each pair of elements is incomparable. The height of a poset is the maximum cardinality of a chain and the width is the maximum cardinality of an anti-chain. We say that y covers x in P, if $x \rightarrow y$ and there is nothing in between, i.e. there is no z such that $x \rightarrow z$ and $z \rightarrow y$ [41].

3.5.6. Hasse Diagram

The cover graph associated with P = (X,P) is the graph G = (X, E) where the edge set E consists of pairs (x,y) for which $x \rightarrow y$ in P. The Hasse diagram is a graph representation of a partially ordered set P if x is lower in the plane than y whenever $x \rightarrow y$.

For example Figure 19 shows the Hasse diagram for P=({1,2,3,4,6,8}, divisibility).

To create the Hasse diagram we follow four easy steps:

1) Construct a digraph representation of the poset where all the arcs point up
2) Eliminate all the loops
3) Eliminate all redundant arcs (transitivity)
4) Eliminate the arrows at the end of each arc (everything points up)

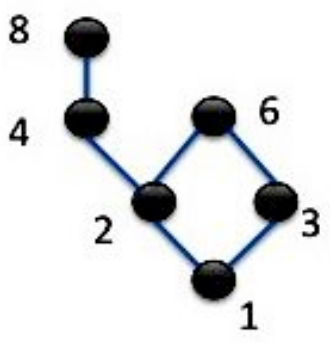


Figure 19: Hasse diagram for P=({1,2,3,4,6,8}, divisibility)

If x→y in the poset, then the point corresponding to x appears lower than the point corresponding to y. An arc between the two points is represented if and only if x covers y or y covers x.

The power set of any set S, $\wp(S)$, is the set of all subsets of S including the empty set and S itself. Note that if S contains exactly s elements, then the cardinality of $\wp(S)$ is 2^s.

If S is the set {teacher.id, users.user_id, teacher.teacher_id}, then the subsets of S are:

{}, {teacher.id}, {users.user_id}, {teacher.teacher_id}, {teacher.id, users.user_id}, {teacher.id, teacher.teacher_id}, {users.user_id, teacher.teacher_id}, {teacher.id, users.user_id, teacher.teacher_id}

and $\wp(S)$ = {{}, {teacher.id}, {users.user_id}, {teacher.teacher_id}, {teacher.id, users.user_id}, {teacher.id, teacher.teacher_id}, {users.user_id, teacher.teacher_id}, {teacher.id, users.user_id, teacher.teacher_id}}.

Figure 20 shows the Hasse diagram of P=($\wp(S)$, ⊆) where ⊆ represents the partial order.

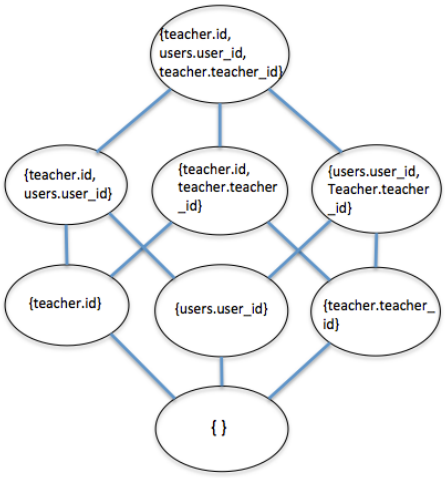


Figure 20: Hasse diagram of P=(℘ (S), ⊆)

P={{}, {teacher.id}, {users.user_id}, {teacher.teacher_id}, {teacher.id, users.user_id}, {teacher.id, teacher.teacher_id}, {users.user_id, teacher.teacher_id}, {teacher.id, users.user_id, teacher.teacher_id}, ⊆)

3.5.7. Maximal Element

If P = (X,P) is a poset, an element x∈X is called a maximal element if there is no y∈X for which x→y. The set of maximal elements is represented by MAX(X,P). This is always an antichain since if there are two elements that are comparable then one of them is not maximal [41]. However, MAX(X,P) may not be as large as the width of (X,P) (the maximum cardinality of an antichain). A maximal element of a poset has no upward line in the Hasse diagram. There may be zero, one, or many elements. Finite posets must have at least one maximal element.

In Figure 20 {teacher.id, users.user_id, teacher.teacher_id} is the maximal element since there is no further element above it. We draw the Hasse diagram of a finite poset in such way, if y covers x, then the point that represents y is higher than the point of x in the

plane. No arrows required in the drawing (directions of the arrows are implicit).

3.5.8. HP Key Search

As a first step we build up a data matrix based on the query templates. The matrix contains all table.key entries for all queries including the query conditions. If a query template has no "WHERE" condition or falls into our limitations then we remove all entries of the involved tables from the matrix. For example, "*SELECT * FROM A*" would not give us a partitioning key for table A since there is no specified key in the query. After building the matrix we create an object key set that includes all the possible TABEL.KEY entries. If we would like to know weather TABLE1.KEY1≤ TABLE2.KEY2 we have to check all queries for TABLE1.KEY1 and TABLE2.KEY2. After this step we find the maximal elements of the object key applying horizontal partitioning rules set. As the last step we compare the maximum elements for partition key(s) selection. For example, consider the queries "*SELECT A.K from B inner JOIN A ON B.K = A.K WHERE A.K = 12*" and "*SELECT * from A WHERE A.K = 54*" and "*SELECT C.K from B inner JOIN C ON C.K = B.K WHERE C.K= 34*" where B.K, A.K, and C.K have one-to-one relationship between each other. Let's build the data matrix D:

	Q1	**Q2**	**Q3**
A.K	1	1	0
B.K	1	0	1
C.K	0	0	1

Create the object key X={A.K, B.K, C.K} and let's draw the Hasse diagram for P=({A.K, B.K, C.K}, HP). Since A.K, B.K, and C.K have one-to-one relationship with each other therefore, they fulfill our requirement for horizontal partitioning.

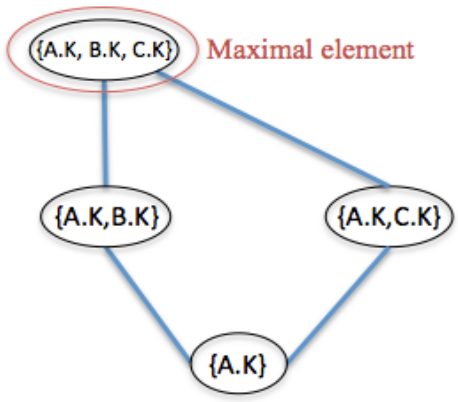


Figure 21: Hasse diagram for P=({A.K, B.K, C.K}, HP)

There is an easy $\binom{n}{2}$ =O(n^2) time algorithm finding MAX(X,P). For each x∈X we check all the pairs (x,y), y∈X. Those elements x which never "loose", i.e. always either y≤x or (x,y) are incomparable, are the maximal elements. In the Hasse diagram (see Figure 21) these are the elements that there is no element covering them. Note that it is not hard to see that in the worst case, if we have no information on the poset, then we need this many comparisons. Indeed, suppose that our algorithm does not compare the pair (x_1, x_2). Then consider two posets (X, P_1) and (X, P_2), where in P_1 no two elements are comparable and in P_2 no two elements are comparable except $x_1 \leq x_2$. Then indeed our algorithm is going to give incorrectly the same result on the two posets. However, note that if we have more information on the poset, then we might be able to do better. For example, if it is a

linear order, then it is well known that we can find the maximum element in *n-1* comparisons. However, here we use the fact that it is a linear order which might not be true for us. Let us also mention here that if we need a little more information on the structure of the poset, then we can apply the following fundamental theorem of posets:

Theorem (Dilworth, 1950, see [41]). *If (X,P) is a poset of width n, then there exists a partition* $X = C_1 \cup C_2 \cup \ldots \cup C_n$ *where each* C_i *is a chain.*

In this case we can partition all tables based on key A.K, B.K, or C.K. Figure 22 shows the algorithm.

algorithm: HP_Key_Search(T, DBR, HP)
input: all existing query templates, database table relationships, horizontal partitioning rules
output: Possible partitioning key(s)

1. Build up data matrix D removing all instances of tables with limitations or without "WHERE" conditions based on query templates
2. Build up Object key X that includes all possible TABLE.KEY combinations
3. Find MAX(X, {P, HP|DBR}): For each x∈X check all the pairs (x,y), y∈X and check for the possible HP conditions
4. Compare maximal elements for key selection and return partitioning key(s)

Figure 22: HP key search algorithm

3.6. Vertical Partitioning

Vertical partitioning splits the table into smaller ones with the same number of rows but fewer columns. We do not require that one can recreate a row of a column in an original format and we do not add an extra ID column of the table in each vertical partition. The

vertically partitioned table contains a subset of the original table. It is a reasonable approach because the queries can be executed on a subset of the original table, thus generating a smaller number of page accesses [63]. The main goal of this partitioning activity is to find sets of columns that are accessed by the query templates. We do not have exactly the same restrictions that horizontal partitioning has. In the case of vertical partitioning, we only need to identify the accessed columns for each table in the query templates.

3.6.1. Operator and Framework Limitations

The parser component of our framework does not have support for queries in the format of an implicit JOIN: "*SELECT * from user, progresses WHERE user.id=progresses.id*" and for three/multiple way joins in the format of "*SELECT user.ID, progresses.ID, teacher.ID FROM user u, progresses p, teacher t WHERE u.id = p.id and p.id = t.id*"**.** Also, we do not parse the following **aggregate functions and Mathematical operators:** SQRT, POWER, EXP, and COUNT(*).

The system can be easily extended for such supports.

Views: Systems with views (virtual subset of a table) are handled as normal table by our algorithm.

Nested queries: We do not support nested queries.

Transactions: We do not support transactions.

Union: We do not support the UNION operator.

3.6.2. VP Key Search

As a first step we build up a data matrix for each table based on the query templates. The matrix contains all key entries of a table for all queries including the query conditions. For example, "*SELECT A.K1, A.K2 FROM A*" would give us K1 and K2 keys to vertically partition table A. After building the matrixes we create object key set for each table that include all the possible KEY entries. As a last step we find the maximal elements of these object keys. For example, consider the queries "*SELECT A.K1 from B inner JOIN A ON B.K1 = A.K2 WHERE A.K2 = 12*" and "*SELECT K2 from A WHERE A.K3 = 54*" and "*SELECT C.K1 from B inner JOIN C ON C.K1 = B.K1 WHERE C.K1= 34*. Let's build the data matrixes D_1, D_2, and D_3 for each table (A, B, C):

A	**Q1**	**Q2**	**Q3**
K1	1	0	0
K2	1	1	0
K3	0	1	0

B	**Q1**	**Q2**	**Q3**
K1	1	0	1

C	**Q1**	**Q2**	**Q3**
K1	0	0	1

Create the key sets (X_1, X_2, and X_3) for each table: $X_1=\{K_1, K_2, K_3\}$, $X_2=\{K_1\}$, and $X_3=\{K_1\}$.

If X_I is the set then the subsets of X_I are:

$\{\}, \{K_1\}, \{K_2\}, \{K_3\}, \{K_1, K_2\}, \{K_1, K_3\}, \{K_2, K_3\}, \{K_1, K_2, K_3\}$

and $\wp(X_I) = \{\{\}, \{K_1\}, \{K_2\}, \{K_3\}, \{K_1, K_2\}, \{K_1, K_3\}, \{K_2, K_3\}, \{K_1, K_2, K_3\}\}$.

Similarly, $\wp(X_2) = \{\{\}, \{K_1\}\}$ and $\wp(X_3) = \{\{\}, \{K_1\}\}$.

Figure 23 shows the Hasse diagram of $P_1=(\wp(X_1), \subseteq)$, $P_2=(\wp(X_2), \subseteq)$, and $P_3=(\wp(X_3), \subseteq)$ where $\subseteq$ represents the partial order.

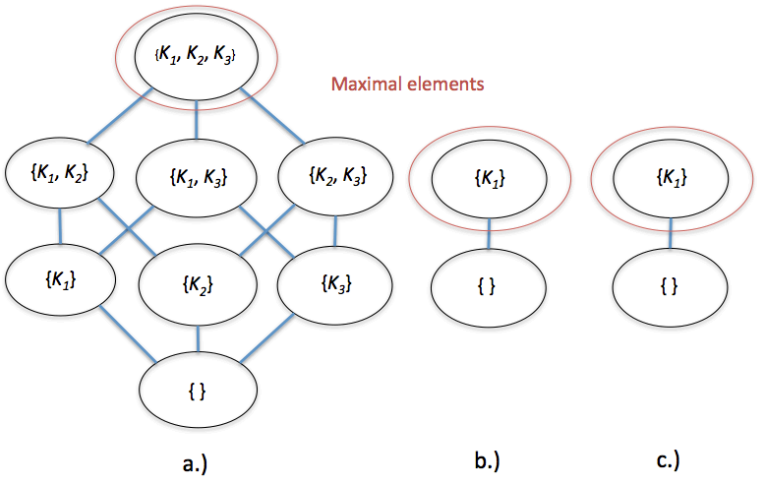


Figure 23: Hasse diagram of P_1(a), P_2(b), and P_3(c)

As mentioned above there is an easy $\binom{n}{2}$ =O(n^2) time algorithm finding MAX(X,P). For each x∈X we check all the pairs (x,y), y∈X. These elements x which never "loose", i.e. always either y≤x or (x,y) are incomparable, are the maximal elements. In the Hasse diagram these are the elements that there is no element covering them. We calculate MAX(X,P) for each table. As an additional step, we check if the total number of candidates in the result sets is not equal to the total number of related table's columns. If yes, then we do not consider the table for vertical partitioning. As a result of this example, we can vertically partition table A based on key A.K1, A.K2, and A.K3, table B based on B.K1, and table C based on C.K1. Figure 24 shows the algorithm.

```
algorithm: VP_Key_Search(T, VP)
input: all existing query templates, vertical partitioning rules
output: Possible partitioning key(s) for each table
For each existing table t
    1. Build up data matrix D removing all instances of tables with
    limitations or with "*" conditions
    2. Build up Object key X that includes all possible KEY
    combinations
    3. Find MAX(X, {P, ⊆}): For each x∈X check all the pairs (x,y),
    y∈X
    4. table.[t] = result
return table.[t]
```

Figure 24: VP key search algorithm

3.7. Combined Vertical Partitioning

Normal vertical partitioning has a drawback. We have to scan all the query templates and find all the referenced columns for a particular table. If one query template references three columns of the table and a second query retrieves all the columns then normal vertical partitioning will not apply the operator on the table. For example, "*SELECT K1, K2, K3 from A*" and "*SELECT * from A*". In this case, table A will not be considered for vertical partitioning. What if we combine vertical partitioning with full replication operator? We can still create a vertically partitioned table A' that involves K1, K2, and K3 columns if we keep a full copy of the original table A as well. This solution combines the two operators. In this case, the first query is answered by using table A' (table A' will be fully replicated across all the nodes as well) and the second query is routed to table A. This solution involves the inserts of the UDI queries into A and A' tables but opens more possibilities for vertical partitioning (see Figure 25).

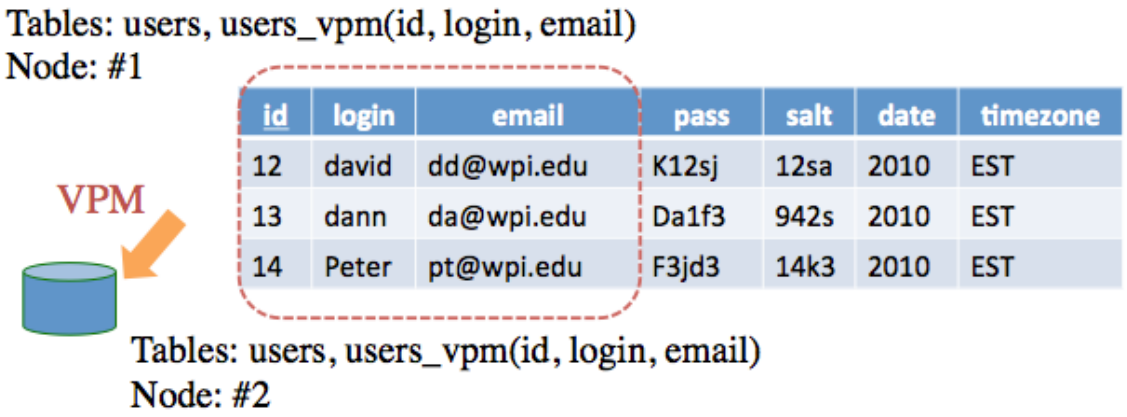


Figure 25: Combined Vertical Partitioning (VPM)

3.7.1. Operator and Framework Limitations

The parser component of our framework does not have support for queries in the format of an implicit JOIN: "*SELECT * from user, progresses WHERE user.id=progresses.id*" and for three/multiple way joins in the format of "*SELECT user.ID, progresses.ID, teacher.ID FROM user u, progresses p, teacher t WHERE u.id = p.id and p.id = t.id*"**.** Also, we do not parse the following **aggregate functions and Mathematical operators:** SQRT, POWER, EXP, and COUNT(*).

The system can be easily extended for such supports.

Views: Systems with views (virtual subset of a table) are handled as normal table by our algorithm

Nested queries: We do not support nested queries.

Transactions: We do not support transactions

Union: We do not support the UNION operator.

3.7.2. VP Combined Key Search

As a first step we build up a data matrix for each table based on the query templates. The matrix contains all key entries of a table for all queries including the query conditions. For example, "*SELECT A.K1, A.K2 FROM A*" would give us K1 and K2 keys to vertically partition table A. After building the matrixes we create object key set for each table that include all set of KEY entries per queries. Additionally to this step, we search for tables that should be kept fully replicated. For example, consider the queries "*SELECT A.K1 from B inner JOIN A ON B.K1 = A.K2 WHERE A.K2 = 12*" and "*SELECT K2 from A WHERE A.K3 = 54*" and "*SELECT * from A"*. Let's build the data matrixes D_1 and D_2 for each table (A, B):

A	Q1	Q2	Q3
K1	**1**	**0**	**0**
K2	**1**	**1**	**0**
K3	**0**	**1**	**0**
*	**0**	**0**	**1**

B	Q1	Q2	Q3
K1	**1**	**0**	**0**

Create the key sets for each table: $X_1=\{K_1, K_2\}$, $X_1=\{K_2, K_3\}$, $X_2=\{K_1\}$, and find table(s) that should be fully replicated.

To be able to answer each query correctly, table A should be fully replicated in partitioning case. Figure 26 shows the algorithm.

```
algorithm: VP_Combined_Key_Search(T, VPM)
input: all existing query templates, combined vertical partitioning
rules
output: Possible partitioning key(s) for each VPM table and
required table(s) to be fully replicated
For each existing table t
      1. Build up data matrix D removing all instances of tables with
      limitations
      2. Build up Object keys that includes all set of KEY entries per
      queries
      3. Find tables that should be kept fully replicated
      4. table.[t] =  Object key sets  + tables to be fully replicated
return table.[t]
```

Figure 26: Combined VP key search algorithm

3.8. Denormalization

In denormalization, one moves from higher to lower normal forms in the database modeling and add redundant data. The performance improvement is achieved because some joins are already pre-computed and the query response time will be minimized [64,65]. For improving the performance of database systems, denormalization has been studied in several projects [66,67,68]. During the normalization task one decomposes tables into smaller ones. One of the main purposes of denormalization is to decrease the number of tables that must be accessed to answer a query. The more tables we have, the more joins we have to perform in the query templates. During the denormalization task one combines tables together to form a bigger one. As the result of denormalization, the data is presented in the same table and there is no need for any previous joins. If one denormalizes two tables with one-to-many relationship, one has the options of how to limit the 'many' relationship in the denormalized table. Either the DBA has pre-knowledge which data segment is used or the created table will end up with more rows

and columns. The denormalized table contains no redundant columns to match the join criteria. We denormalize tables based on the join frequency in the query templates. In default, the denormalization ratio is set to 10%. That means the occurrence of the same join is equal or greater than 10%. If the condition is fulfilled, we denormalize the involved tables based on their join condition.

3.8.1. Operator and Framework Limitations

The parser component of our framework does not have support for queries in the format of an implicit JOIN: "*SELECT * from user, progresses WHERE user.id=progresses.id*" and for three/multiple way joins in the format of "*SELECT user.ID, progresses.ID, teacher.ID FROM user u, progresses p, teacher t WHERE u.id = p.id and p.id = t.id*"**.** Also, we do not parse the following **aggregate functions and Mathematical operators:** SQRT, POWER, EXP, and COUNT(*).

The system can be easily extended for such supports.

Views: Systems with views (virtual subset of a table) are handled as normal table by our algorithm

Nested queries: We do not support nested queries.

Transactions: We do not support transactions

Union: We do not support the UNION operator.

The next chapter will talk about the framework and its performance evaluation.

3.9. Conclusion

In this chapter, we studied the problem of scalability in web applications; in specific we considered distributing load across multiple database servers. The contribution of this chapter is a methodology, which can help Web-based applications that deal with scalability problems and helps to figure out what is a good database layout for a particular Web-based application given a query workload. The typical scaling bottleneck of an application is at the database side.

We proposed a data placement algorithm that considers a data placement technique and determines the best possible layout of tables across multiple database servers for a given query workload. This layout algorithm is capable of determining a possible data placement based on the query templates, constraints, and the optimization goal using four operators (full replication, horizontal partitioning, vertical partitioning, denormalization) and arbitrary number of database servers answering each query by a single node.

We introduced a state space search methodology by which we search for better database layouts. By conceptualizing the problem as a state space search problem and by doing a full state space search, we were able to physically create the layouts and evaluate the overall response time of the system to parameterize the guiding rules. One of the assumptions we actually impose upon ourselves is that queries should be answerable by a single database node. By making this assumption we simplified the processing of individual queries to the databases. Of course, sometimes DBAs want to write queries that can go across all database nodes involving multiple tables, e.g. for analytical

purposes but these analytic queries can be executed as a background task by the DBAs. In this chapter, we applied posset theory and its Hasse diagram representation to algorithmically describe our operators and visualize their operations. We propose to identify other features that can be important to guide the search and use them as an input of a machine learning technique to create general rules about when to use the different layout operators. We would like to use our middleware to actually find more interesting rules involving different features. We propose to use them as rules of thumb of a machine learning system to recommend when to use the different layout operators.

The next chapter will talk about the architecture of the framework, implemented components, and it introduces a complete workflow of the system.

4. The Framework

Figure 27 illustrates the high-level architecture of the system and its main inputs. The workload specifies the sequence of queries that were collected during normal usage of the database. It needs to contain all incoming query templates of the application. Constraints represent the relationships between tables. The placement algorithm determines the applicable operators, partitions, and partitioning keys based on the workload and constraints. The system also considers the application source database and the available database nodes for partitioning which applies the nodes' addresses and database connector strings.

Figure 28 shows the modularized architecture of the system. The Data Placement Algorithm (DPA) is responsible for determining the valid sets of tables and keys for each operator based on the given workload, constraints, and node information.

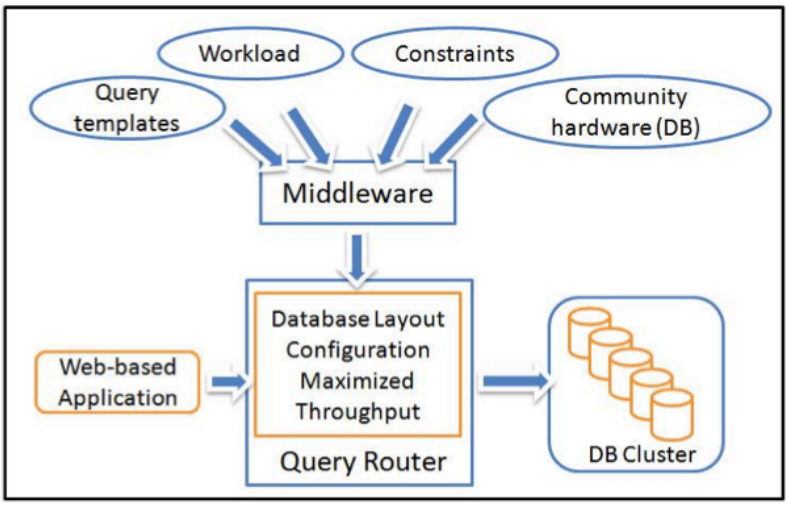


Figure 27: High-level architecture of the framework

Appendix J shows a flowchart of step 4 of the DPA (how the operator-key pair is selected) and Appendix K describes a flowchart of step 6.1 of the DPA (how the partitioning key is selected). As soon as the algorithm creates the sets it passes them to the State Space Search Module (SSSM). This module is the heart of the search. If the SSSM determines a valid state then it contacts the Layout Module (LM) to initiate the creation of the physical configuration with the selected operator. The LM stores information about the created configurations in the Layout Bank (LB). Therefore, it asks the LB for the required configuration. If the LB has no previous information about the requested layout then the LM starts the layout generation process. First, it connects to the source database to initialize the layout generation.

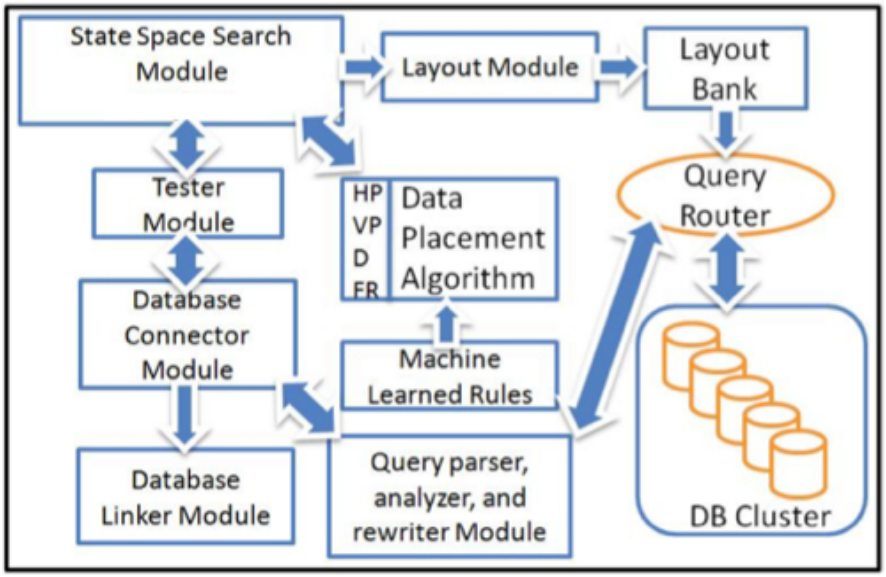


Figure 28: Modularized architecture of the framework

Then it collects information about the source table Users (e.g. column names and types, indices, triggers, etc.) and generates the new table schema (Table name + Applied operator + New Table identifier + Partitioning Key) utilizing the given database nodes

(e.g. Users_HP_2abc4_id). In the next step the LM creates the new tables using the cloned table information on each node. With a pre-defined hash function, it distributes the tuples among multiple database nodes before sending the layout information to the LB. When the layout creation finishes, the SSSM initiates a test request to measure response time of the system. The Tester Module (TM) simulates the real world example with multiple application servers. Each simulated application server uses the workload to generate hundreds or thousands of requests for the back-end. The Database Connector (DCM) and Linker (DLM) modules are responsible for initializing and maintaining the database connections towards the available database nodes. The Query Analyzer Module (QAM) parses the queries in the workload and rewrites them to replace the original table names with the partitioned ones (e.g. replace table Users with Users_HP_2abc4_id). It uses the same hash function the LM applies to determine the correct database nodes for data retrieval. Once the data is laid out on the database servers, the LM updates the middleware's Query Router (QR) about the new changes in the layout configuration. The QR maintains multiple connections to the database nodes, routes queries to the correct node, and transfers results back to the requestor. As soon as a new configuration is laid out, the Web-based application can connect to the QR without needing any code modification. The application detects the QR as a database node that manages and hides the configuration differences utilizing multiple databases.

4.1. Routing the Queries

After the data is laid out across the different database servers, the system is ready to start processing the queries. The query router routes the queries to the appropriate database

server/(s): a select query is sent to the appropriate database server, and an UDI query is sent to all the appropriate database servers. For performing the routing, the query router utilizes the dataLayout that is returned by the data placement algorithm. In addition to routing the queries correctly, the query router must also ensure that the database servers are utilized effectively. For this, we need to be executing multiple queries on any database server at any instant, while also maintaining the correct in-order semantics specified by the application. Our solution includes an efficient query router that maintains multiple connections for each database server, thus enabling multiple concurrent queries on a database server. Our detailed architecture for routing queries is shown in Figure 29.

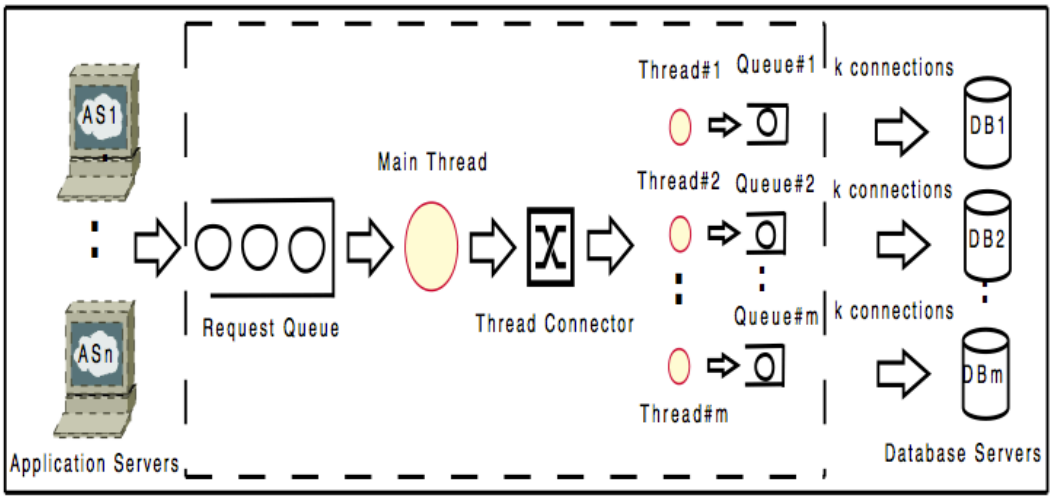


Figure 29: Architecture of the Query Router

The thread that handles the requests for a database server maintains a queue of requests that the server needs to process, multiple connections to the server for executing multiple queries concurrently. The queries from all the application servers are sent to the query router, where the requests are queued. The query router also maintains how the tables are placed on the different database server nodes (using the dataLayout structure returned by

the placement algorithm); this information is used to route a query to the appropriate database server node/(s). In our system, how to route a query is determined statically and does not vary based on the current load on the database servers. A select query is routed by the query router to one database server, whereas an update query is routed to all of the appropriate database servers. For example, a query of type Q1 (see Table 1) may be routed to node 1; a query of type Q2 may be routed to node 5; a query of type Q6 has to be routed to all the five nodes.

For replicated tables when a query can be answered by more than one node, our system routes the queries in a simple round-robin fashion. This routing ensures that the database servers are equally loaded. Each database server is managed by a thread that maintains two data structures: a queue of requests it has received, and a lock table to handle conflicting select and UDI queries. In order to increase the performance of each database server, the thread for the database server maintains multiple connections to that server; thus multiple queries can be executed simultaneously on a single server. If multiple queries can be scheduled simultaneously on a database server, we need to implement a simple locking mechanism. Let us illustrate how the locking mechanism is implemented in our system using a lock table. Consider queries of type Q4 and Q7 (see table 1) that are conflicting: Q4 reads from the users table while Q7 inserts into the users table. If there is a query of type Q4 and a query of type Q7 both waiting to be serviced in that order, they cannot be scheduled simultaneously. Rather, we have to wait for Q4 to be finished before Q7 is scheduled. We cannot let the database server handle the conflict management, because it will not guarantee the serial order of Q4 and Q7. Such conflicts are handled using the lock table as follows: first the thread for the database server

examines the current query and sees if it can obtain the appropriate locks (read/exclusive lock). If the locks are available, then the query is scheduled on one of the available connections; otherwise, it waits till the lock is available and then the query is scheduled on one of the connections. When the query is finished, the locks are updated accordingly. While a query is waiting for a lock to be available, the following queries in the thread queue are not scheduled (even though locks may be available for those queries); this solution is done for simplifying our architecture.

4.2. The Tester Module

The tester module is responsible for measuring the total system response time using the workload. The module models the expected usage of the application by simulating users who can access the application at the same time. This modeling makes the measurements more realistic to the real word application. There is an option to set the standard deviation (stderr) requirement between the results of each measurement. In default, this value is 0.05. This means the tester module continues to determine the system response time until the standard deviation of the measurements' results is less than or equal to 0.05. This measurement method also eliminates the cache warm-up effects so that the disk caches get populated with valid data and it makes the measurement accurate. The module also handles the database cleaning. The workload can contain insert/delete queries that modify the total number of tuples in a table after inserting or deleting rows. After each measurement, it runs the cleaner workload to delete or insert into the tables and clean the previously inserted or deleted extra tuples.

4.3. The workflow

The framework is automatically decides on the suitable partitioning strategy without any human intervention. TPC-W [43], the Industry Standard eBusiness transactional web benchmark's tables and query templates were used to generate different database configurations. Appendix C shows the query templates of the Java TPC-W [43] Implementation distribution (PHARM University of Wisconsin – Madison). This Web-based application has 48 query templates.

The start condition requires four inputs. The first input is the query template file that includes all the possible mappings of the logical model to SQL statements (see appendix C for TPC-W). The second input is the workload file that contains the actual percentage of queries of each template that the application typically processes. The system can have more than one workload descriptor as input based on the measurement needs. If one would like to increase the number of simulated users who can access the application at the same time then the framework requires as many workload files as the simulated users. The third input is the constraint descriptor file that describes the relationships between tables and columns (see Appendix E for the constraint file of TPC-W). The last input is the database node descriptor file that describes the server names and connection details (database name, port, user, and password). The framework starts to analyze the query templates and determines the partitioning possibilities. After analyzing them it starts the state space search over layouts. The state space search module conducts the full state based search. When a state is created, the framework physically lays out the data, updates the query router with the actual layout configuration, and measures the total system

response time using the given workload. Furthermore, it conducts the significance testing of the results with a significance level of 0.05. During the search a result file is generated. The result file contains the details of the measurements (total system response time of each state and the related layout configuration) and gives information about the best path. Figure 30 shows the configuration of the measurements.

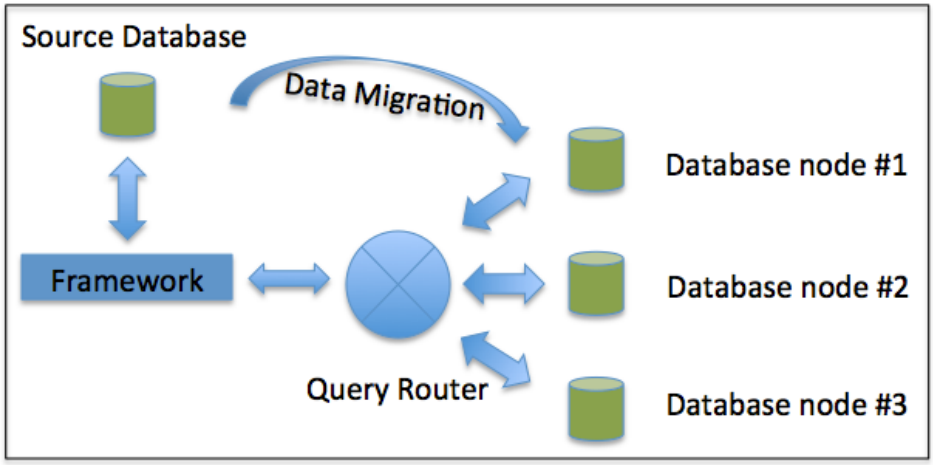


Figure 30: Configuration of the measurements

Table 2 displays the details of the server infrastructure.

Table 2: Information of the infrastructure

Server	Server function	Information	Database
1	Framework	Intel Pentium 4, 3.0Ghz CPU, 4 GB RAM, 64 bit Ubuntu 4.1.2	-
2	Query Router	Intel Xeon 2 core 3.0Ghz CPU, 4 GB RAM, 64 bit SUSE 11.1	-
3	Source Database	Intel Xeon 4 core 3.2Ghz CPU, 8 GB RAM, 64 bit Debian 3.4.3 Sarge	PostgreSQL 8.2
4	Database Node 1	Intel Xeon 4 core 3.2Ghz CPU, 8 GB RAM, 64 bit FreeBSD 7.3	PostgreSQL 8.2
5	Database Node 2	Intel Xeon 4 core 3.2Ghz CPU, 4 GB RAM, 64 bit FreeBSD 8.2	PostgreSQL 8.2
6	Database Node 3	Intel Xeon 4 core 3.2Ghz CPU, 4 GB RAM, 32 bit FreeBSD 7.2	PostgreSQL 8.2

As an example, table 3 presents a result of the state based search for TPC-W.

Table 3: The results of the state based search for TPC-W

State	Operator	Table	Improvement State 0	Improvement Parent State	System Response Time	TTEST State 0	TTEST Parent	Parent
0	FR	ALL	N/A	N/A	8.919s	N/A	N/A	0
1	HP	cc_xacts Key: cx_o_id New table: cc_xacts_HP_2af5f4b9	YES (0.267s)	YES (0.267s)	8.652s	Significantly better	Significantly better	0
2	HP	shopping_cart_line Key: scl_sc_id New table: shopping_cart_line_HP_461bacd1	YES (0.55s)	YES (0.283s)	8.369s	Significantly better	Significantly better	1
3	DN	shopping_cart_line, item New table: shopping_cart_line_item_DN_78a96b32	YES (0.517s)	YES (0.25s)	8.402s	Significantly better	Significantly better	1
4	HP	shopping_cart_line Key: scl_sc_id New table: shopping_cart_line_HP_461bacd1	YES (0.589s)	YES (0.04s)	8.330s	Significantly better	Significantly better	0
5	HP	cc_xacts Key: cx_o_id New table: cc_xacts_HP_2af5f4b9	YES (0.638s)	YES (0.05s)	8.281s	Significantly better	Significantly better	4
6	VP	cc_xacts Keys: cx_o_id, cx_type, cx_num, cx_name, cx_expire, cx_auth_id, cx_xact_amt, cx_xact_date New table: cc_xacts_VP_973d6367	YES (0.566s)	NO (0.02s)	8.353s	Significantly better	Not significantly worse	4
7	VP	cc_xacts Keys: cx_o_id, cx_type, cx_num, cx_name, cx_expire, cx_auth_id, cx_xact_amt, cx_xact_date New table: cc_xacts_VP_973d6367	NO (0.02s)	NO (0.02s)	8.9s	Not significantly better	Not significantly better	0

8	DN	shopping_cart_line, item New table: shopping_cart_line_item _DN_78a96b32	YES (0.337s)	YES (0.337s)	8.582s	Significantly better	Significantly better	0
9	HP	cc_xacts Key: cx_o_id New table: cc_xacts_HP_2af5f4b9	YES (0.504s)	YES (0.167s)	8.415s	Significantly better	Significantly better	8
10	VP	cc_xacts Keys: cx_o_id, cx_type, cx_num, cx_name, cx_expire, cx_auth_id, cx_xact_amt, cx_xact_date New table: cc_xacts_VP_973d6367	YES (0.503s)	YES (0.166s)	8.416s	Significantly better	Significantly better	8

The cardinality of the item table, TPC-W's scaling factor was one million rows and the workload contained 144 queries, 3 times all the query templates. We started with two simultaneous threads for the measurements (Emulated Browsers or EBs). The standard error between the threads' results was less than or equal to 0.05. Path[0-4-5] minimizes the total system response time the most: shopping_cart_line and cc_xacts tables are horizontally partitioned, and all the other tables are fully replicated. Compared to state 0 (8.919s) we minimized the total system response time by 7.153% (0.638s). Figure 31 visualize the complete state space search.

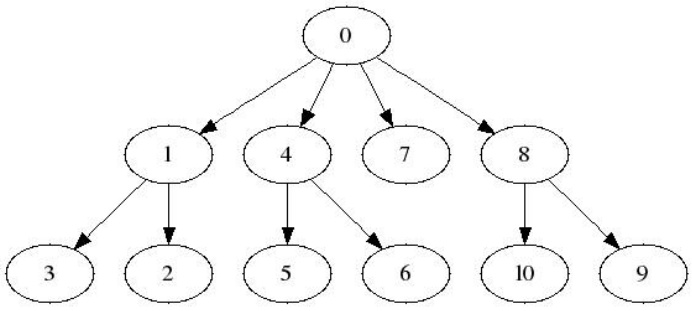


Figure 31: TPC-W state space search

We used the determined best layout configuration (Path[0-4-5]) to repeat the measurement with 100 EBs. We scaled the tables (number of rows) according to the cardinality of the various database tables as a function of number of EBs [76] (see Appendix F). For 100 EBs the total system response time was 28.442s using Path[0-4-5] and 47.899s for State 0 (all tables are fully replicated). The standard deviation was 0.28. The t-test result showed significant improvement. We minimized the total system response time by 40% (19.457s). We also captured the CPU and memory utilization of the query router to ensure that the measurement will not consume 100% of the overall CPU and the memory (swap effect). The highest average (all cores) CPU utilization was 50.2% and swap was not being used. The framework CPU and memory utilization is not significant. The framework and the query router are implemented in python 2.6.6. Figure 32 shows the results of the two experiments.

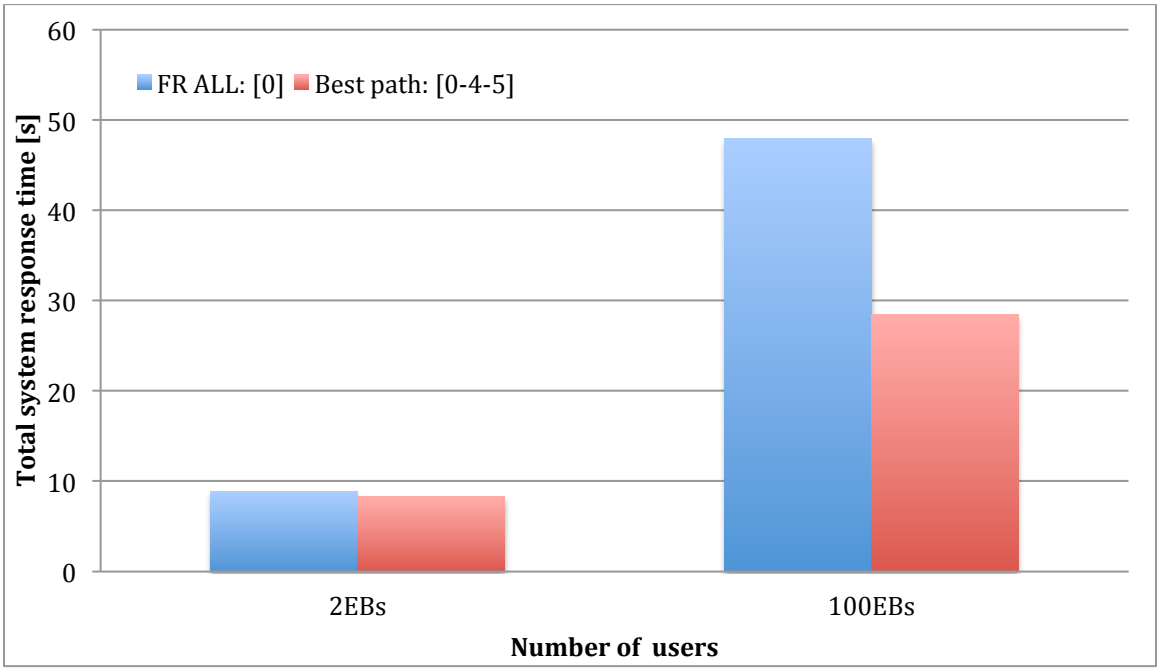


Figure 32: Total system response time vs. number of users (State 0 and best path)

We can see that the cardinality scaling of the database lead to significant improvement and the query router handled multiple clients efficiently.

4.4. Conclusion

The contribution of this chapter is an established middleware that is general and that can be used by any Web-based application. This chapter describes a designed and developed shared-nothing data replication framework for Web-based applications with a state based search component to predict when to choose between horizontal partitioning, vertical partitioning, denormalization or full replication layout operators.

This chapter also introduces its main components. For routing the queries to the appropriate database servers, we developed an efficient query router that is introduced via query routing examples. We also describe our implemented tester module for response time measurements. This chapter establishes empirical results that are described in the workflow section. With our methodology we were able to minimize the total system response time significantly. We report our experimental results -layout operators and best path- using TPC-W and performing the layout search over multiple database nodes with different database cardinalities.

The next chapter will talk about best practices and trade-offs detection to help learn rules and features to input them into a machine learning method.

5. Identification of Trade-Offs

We would like to collect empirical data that can help the state space search to focus on creating layout configurations that could boost the performance of the application. These data are the key to knowing the "ground truth" about what is effective and under what conditions. After collecting empirical data, where these four different operators have been applied, we can use the created configurations as input into a machine learning component to learn rules. These machine learned rules can help to govern the physical design of the database across an arbitrary number of computer nodes. This help, in turn, allows the database placement algorithm to get better over time as its trains over a set of examples.

5.1. Cut-off Points

Cut-off points help the state space search to focus on creating layout configurations that could boost the performance of the application. Cut-off points reduce the size of the search space by eliminating valid table-operator key-pairs because of their possible negative performance effect on the system. To determine these points we turned our attention towards database best practices [37]:

- "horizontal partitioning can increase the performance by splitting large tables into smaller ones. This results in smaller data size and queries can run faster";
- "if a table has many rows, then horizontal partitioning can help to increase the

performance";

- "horizontal partitioning can help to increase the performance if the application is UDI intensive";
- "vertical partitioning can increase the performance by dividing tables into multiple tables that contain fewer columns";
- "vertical partitioning lets queries scan less data and this increases query performance";
- "if a query is focused on a sub-set of columns, then consider vertically partitioning the table";
- "if a table is read intensive, then consider full replication to distribute the load";
- "if the partitioned data is skewed that could effect the performance";
- "if the queries are focused on a specific node in a distributed environment that could effect the performance";
- "if the table is join heavy, then denormalization can help to increase the performance";

These and similar rules (see chapter 6) can only suggest the DBAs what to do but cannot tell them how to partition the database effectively considering the interactions of rules. To explore interactions, guide our search, and to train our machine learning component we created different configurations to find cut-off points. All of our cut-off points form a binary system that reflects when one operator is better than the other one in terms of a suggested feature. In this binary system 1 means that the attribute (e.g. number of columns) is high and 0 means low. For example, can we identify a cut-off point between horizontal and vertical partitioning if a query focuses on a subset of table columns? We

used TPC-W query templates and table schemas to identify 14 cut-off points:

1. Vertical Partitioning vs. Horizontal Partitioning in terms of number of columns;
2. Vertical Partitioning vs. Full Replication in terms of number of columns;
3. Vertical Partitioning vs. Full Replication in terms of UDI vs. read ratio;
4. Denormalization vs. Full Replication in terms of join heaviness;
5. Vertical Partitioning vs. Denormalization in terms of join heaviness;
6. Horizontal Partitioning vs. Denormalization in terms of join heaviness;
7. Vertical Partitioning vs. Denormalization in terms of number of columns;
8. Vertical Partitioning vs. Denormalization in terms of UDI vs. read ratio;
9. Horizontal Partitioning vs. Denormalization in terms of number of rows;
10. Full Replication vs. Denormalization in terms of number of rows;
11. Vertical Partitioning vs. Denormalization in terms of number of rows;
12. Vertical Partitioning vs. Horizontal Partitioning in terms of workload balance;
13. Horizontal Partitioning vs. Full Replication in terms of workload balance;
14. Horizontal Partitioning vs. Denormalization in terms of workload balance;

Join heaviness gives a measure of the join intensity of two tables, workload balance gives a measure of the routing of the queries to the same node in a distributed environment, and skewness gives a measure of asymmetry in the distribution of the data values. We measured each measurement with two threads where the standard error requirement between the results of the threads was 0.05. Figure 33 shows one of the identified cut-off points when vertical partitioning is better than horizontal partitioning. To identify the points we used our framework with different combinations and modifications of TPC-W

query templates (Appendix C) in the workload. We utilized three database nodes and two emulated browsers. The initial cardinality of the TPC-W item table was set for 1M tuples.

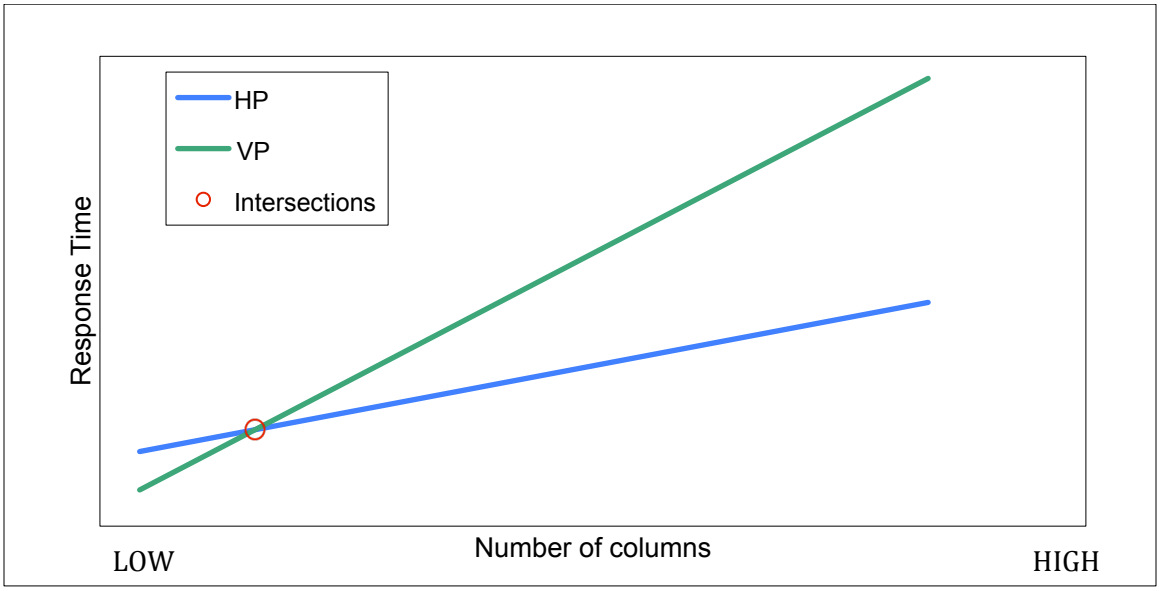


Figure 33: Cut-off point: HP vs. VP (number of columns)
Before the cut-off point vertical partitioning is better than horizontal partitioning and vice-versa after the cut-off point

Figure 33 shows that we horizontally and vertically partitioned the item table with the highest number of columns and changed the number of referenced columns in the query template (Appendix C, query template ID 26) from 1 to the total number of columns-1. The query template frequency was in the range of thousands in the workload. After the change, we re-run the framework to create a new partition and re-measured the total system response time. The red circle shows the cut-off point. Before the cut-off point vertical partitioning is better than horizontal partitioning in terms of number of columns and vice-versa after the cut-off point. We used the same method to determine all 14 cut-off points and we changed different attribute in each case. Figure 34,35, and 36 present the cut-off points.

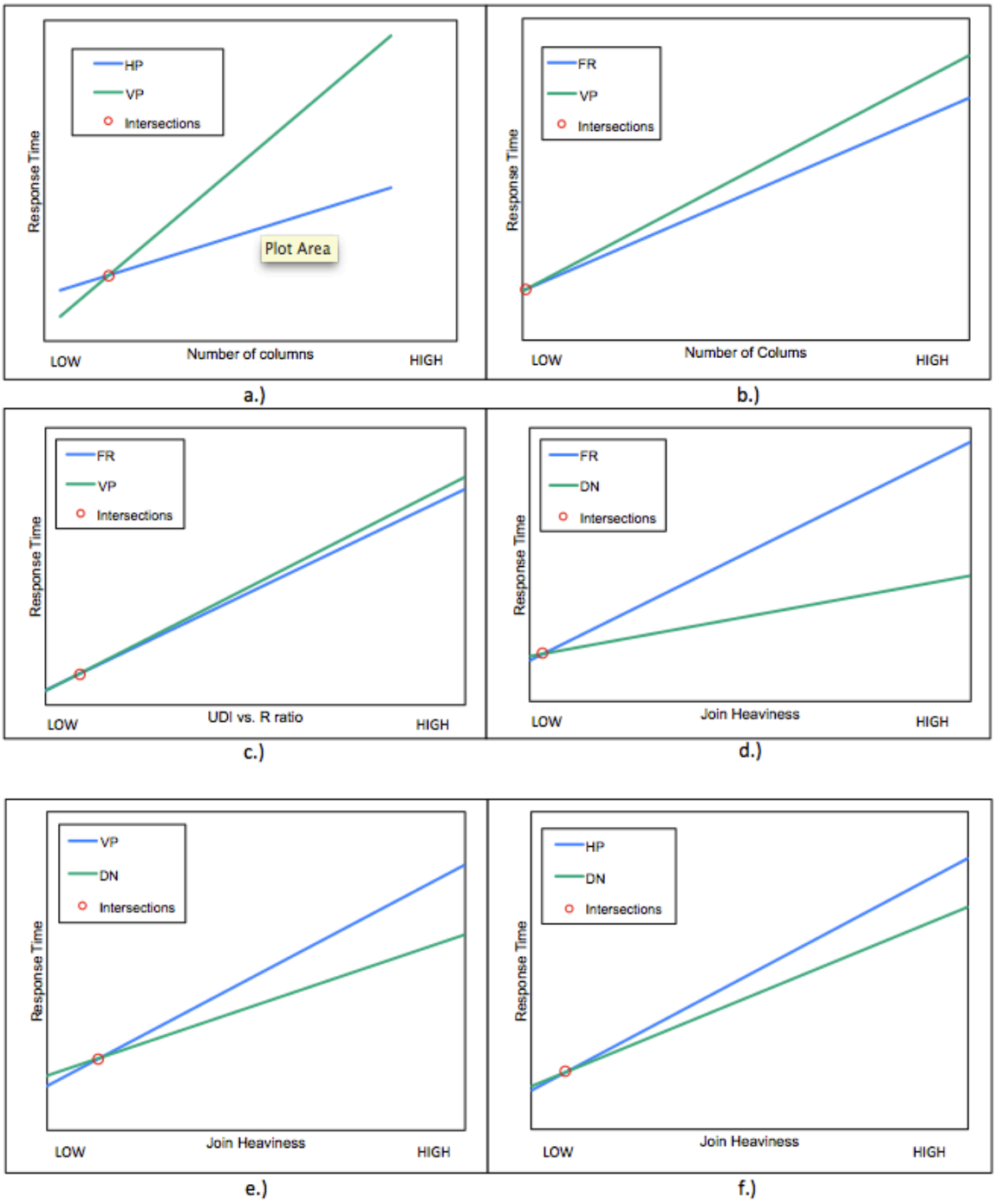


Figure 34: Cut-off points I.

a.) HP vs. VP (#columns): before the cut-off point VP is better than HP and vice-versa after the point
b.) FR vs. VP (#columns): before the cut-off point VP is better than FR and vice-versa after the point
c.) FR vs. VP (UDI vs. Read): before the cut-off point VP is better than FR and vice-versa after the point
d.) FR vs. DN (join heaviness): before the cut-off point FR is better than DN and vice-versa after the point
e.) VP vs. DN (join heaviness): before the cut-off point VP is better than DN and vice-versa after the point
f.) HP vs. DN (join heaviness): before the cut-off point HP is better than DN and vice-versa after the point

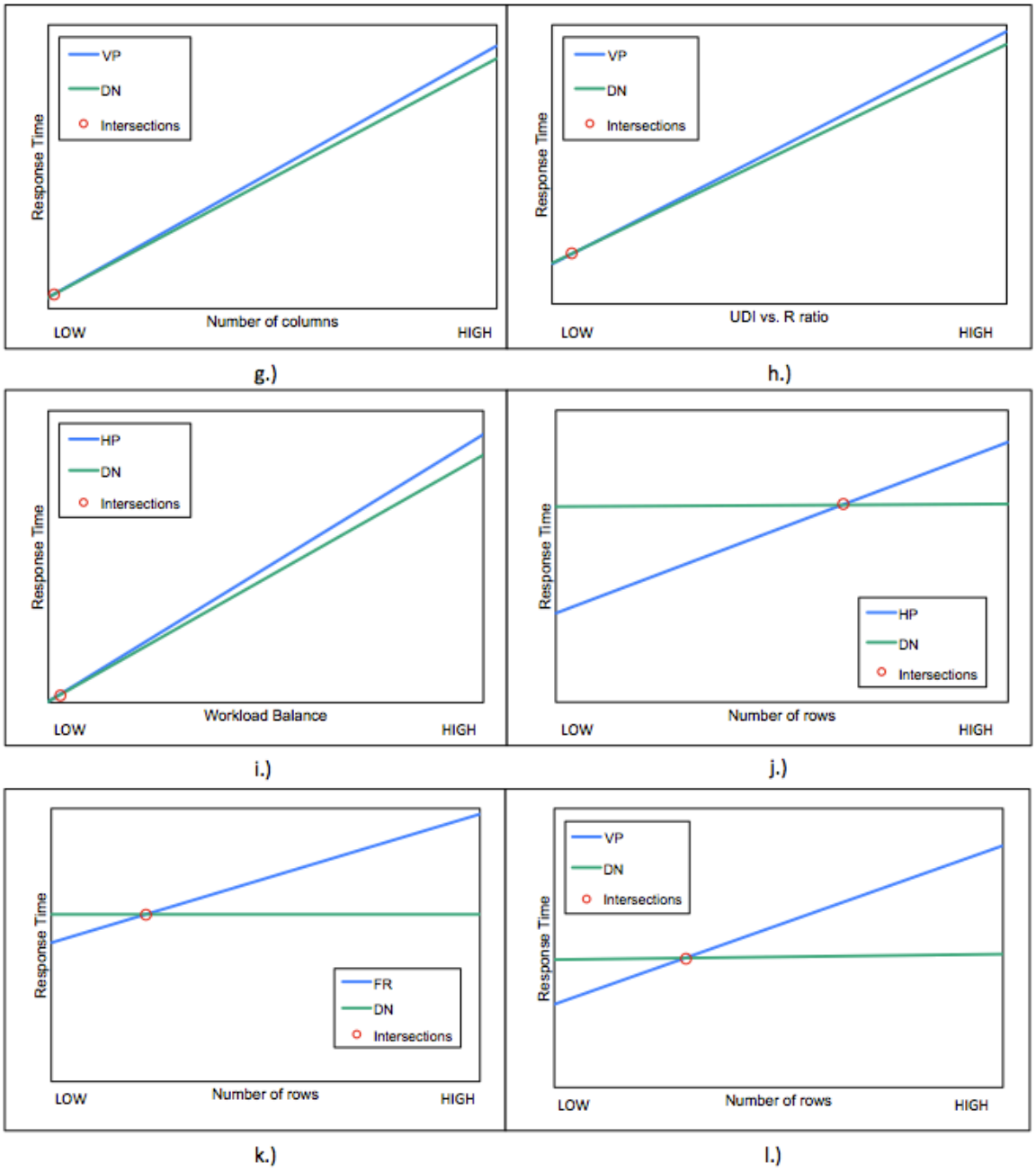
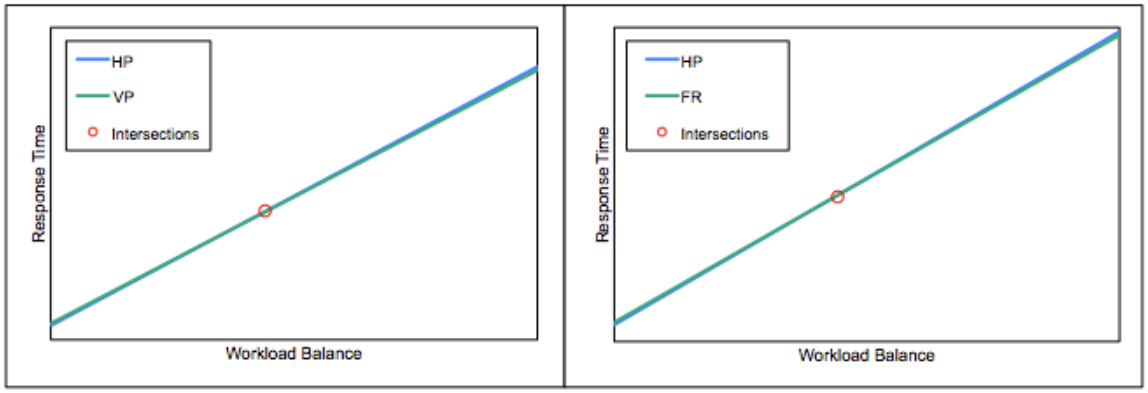


Figure 35: Cut-off points II.

g.) VP vs. DN (#columns): before the cut-off point VP is better than DN and vice-versa after the point
h.) VP vs. DN (UDI vs. R): before the cut-off point VP is better than DN and vice-versa
i.) HP vs. DN (workload): before the cut-off point HP is better than DN and vice-versa after the point
j.) HP vs. DN (#rows): before the cut-off point HP is better than DN and vice-versa after the point
k.) FR vs. DN (#rows): before the cut-off point FR is better than DN and vice-versa after the point
l.) VP vs. DN (#rows): before the cut-off point VP is better than DN and vice-versa after the point



m.) n.)

Figure 36: Cut-off points III.

g.) HP vs. VP (workload): before the cut-off point HP is better than VP and vice-versa after the point
h.) HP vs. FR (workload): before the cut-off point HP is better than FR and vice-versa after the point

5.2. Conclusion

This chapter has a contribution to show how we can go and parameterize database rule of thumbs for cut-off values. Cut-off points help to learn rules that can be effective at speeding up the entire system. By utilizing these cut-off points and their attributes, we can determine possible sets of important features that we need to take into consideration to learn rules with a machine learning method. With the learned rules the state space search can select and focus on creating layout configurations that could boost the performance of the application. This chapter introduced 14 rules and their attributes like "number of rows" or "join heaviness" that we can consider as a set if important features to take into consideration to learn a general rule. The method for parameterizing cut-off rules is quite general and further cut-off point/attribute pairs can be added to the system easily.

The next chapter will describe how we machine learned rules and built our model for operator prediction.

6. Machine Learned Rules

By characterizing the problem as a state space search over database layout configurations, we iteratively minimize the total cost of the workload creating different database layouts and minimize the total system response time. We do a time intensive search over different layouts. There can be complex interactions between the four different operators, which make it even more difficult to predict what the best way to do is. After collecting empirical data where these four different layout operators have been applied, we use the created configurations as input into a machine-learning component, to predict when to use them. This process, in turn, would allow the database placement algorithm to get better over time and reduce the execution time of a long running brute-force search. A natural consequence of making the search more efficient by using machine learning as broadly outlined above would be enhancing the applicability of the system. To test the machine learning idea we gathered data using our framework involving the TPC-W benchmark schemas, and generated machine learned rules. We can then see how good these rules are to search for the optimal layout in similar settings. Our knowledge is based on the different database configurations and schema attributes of TPC-W that we created. [81] proposed a solution to use Bayesian machine learning to find the optimal matching of attributes between two semantically related schemas. They mainly focused on probabilistic matching of domains by DBAs. [82] introduces "discriminators" that could work well to define classifiers. They noted that the "discriminators" should be selected by experiments and that they can be expanded any time on per-DBMS (Database Management Systems) basis (extra discriminators). They propose a Neural Network [83] approach for classification without being given any rule,

but the method has a long training curve and cannot be easily adopted to deal with different database systems. For classification, it is natural to pick up a machine learning method that works to generate rules such as JRip [79] or J.48 [80] given binary attributes. Decision trees have proven to be an efficient way of classification [78]. While it has been seen that Neural Networks are hard to be trained and adopted to a new DBMS [82] Decision Trees could be adopted easily and perform well in the case of small and noisy datasets [77]. We are focused on rule generation as well. Because our instances are describable by attribute-value pairs, the target function is discrete valued (HP, VP, DN, or FR), and the binary cut-off points are determined, we consider the J.48 decision tree for classification. Chapter 6.3 describes the modified Data Placement Algorithm that includes the model prediction step and it also presents the structure of the learned model using unseen data. Of particular interest to us is the fact that a decision tree could be represented as a set of if-else rules based on binary attributes. Like in any supervised learning method, we need a training dataset for learning the decision tree. A training dataset has points that come in feature-label pairs x,y where x is a feature vector and y is the label. Our first task is thus to generate relevant features and their associated labels and to determine the ground truth via extensive measurements on the framework. In the next sub-chapter we describe how these data were collected with a particular emphasis on what were the relevant features and which features were selected.

6.1. Data Collection and Feature Selection

In this chapter we review how the data was collected and how the features were selected.

We also present what we call a “relevance matrix” representing the ground truth and the selected features. The results of the actual measurements are called the ground truth. The relevance matrix also gives an intuitive insight on which operator is more relevant in a given case (given our feature set). For example, feature 1 (Join heaviness: lots of joins per table A in the overall workload compared to other tables) is true and feature 2 (UDI and Read ratio) is false then the relevance matrix indicates that the horizontal partitioning operator minimizes the total system response time the best. It does not just indicate the best operator but also indicates the set of possible operators and how they differ statistically (from each other and the comparing factor that is Full Replication operator) in terms of total system response time. It is important to note that we compared all applicable operators to full replication because of our initial distribution policy that fully replicates all tables across all the nodes (state 0). Cut-off points are manual input to our algorithm based on the determined rules (Chapter 5).

6.1.1. Feature Selection

We parameterize common guidelines that DBAs use as heuristics in doing this task manually. Figure 37 shows a possible group of features that one can apply to parameterize guidelines for the search. The features can be broadly divided into three categories: table-related, query- and workload-related, and state-related features. Table-related features like table size, distinct values, number of columns, etc. can help to pre-select the applicable operators for a specific table. ”Is schema heavy?” feature considers the table schema heavy if the column type can generate high database memory and caching demand. For example, if the column type is text, byte, xml, etc. and the

Table Features	Query & Workload Features
Table size	UDI vs. R query ratio
Primary Key	% of UDI and R queries for Table A
Foreign Key	% of queries of Table A with "WHERE" condition
# of indexes	% of "JOINS" on Table A comparing to other JOIN queries
# of distinct values	% of queries that involves Table A compared to other queries
# of columns	% of queries that involves Table A and other tables compared to all queries
Is schema heavy?	% of queries that involves Table A's columns compared to other queries
# of INSERTS	The frequency of UDI and R queries
# of UPDATES	
# of DELETES	
# of table accesses	

Table Features
Full Replication
Horizontal Partitioning
Vertical Partitioning
Denormalization
Total Throughput

Figure 37: Illustrating Table, Query, Workload, and State features

frequency of the column specific retrieval query is high then one should help the database to share the memory requirements among multiple nodes. Query and workload features are also important to determine exact cut-off points. For example, a table with few tuples is not worth horizontally partition unless the number of table-related UDI vs. read ratio is high enough. As another example: analyze the query templates in the workload and determine a cut-off point to characterize the application as write intensive. Feature selection from the set listed in Figure 37 is an important task, one way to inform ourselves better about the same is to turn to perspectives from database best practices [37] such as:

- "*when the number of update/delete/insert queries on a table is small compared to the number of retrieval queries (e.g selects), then one should fully replicate*";
- "*when the number of update/delete/insert queries on a table is large compared to*

the number of retrieval queries, then one should horizontally partition";

- "*if there is a wide table but a lot of read queries are focused on a small set of columns of the table, then one should vertically partition*";
- "*when the table size is large, then one should horizontally partition*";
- "*when one frequently joins two tables, then one should denormalize them*";
- "*vertical partitioning can reduce the amount of data that needs to be scanned to answer the query*".

Other than ideas from database best practices [37], we used our cut-off rules, recommended experimental evidence [82] and domain knowledge to select a subset of features. These features are listed below:

1. Update/Delete/Insert versus read query ratio of the table;
2. Number of columns of the table;
3. Number of rows in the table;
4. Join heaviness: this feature gives a measure of the join intensity of a table in the overall workload compared to other tables
5. Workload balance: this feature gives a measure of the routing of the queries to the same node in a distributed environment
6. Skewness: this feature gives a measure of asymmetry in the distribution of the data values. If we have 2 database nodes and all the 9,000 records are distributed under the same node after hashing, then this distribution creates a positive skewness (+1) for the table under the node. If the same value is applicable for 100 records only, then this distribution creates a negative skewness (-1) for the same table under the node. Possibly, most of the queries from the workload can access only that specific node after computing the hash function of the table's key e.g. the majority of the logins are from the same school and the data are partitioned based on schools. In this case that node could be a bottleneck on the load based on skewness and the access pattern of the workload.

Except skewness, all features are binary valued. For example, UDI vs. read ratio of a table can be high (1:only UDI queries) or low (0:only read queries). Skewness can take three values (0: no skewness, +1: positive skewness, -1: negative skewness).

6.1.2. The Relevance Matrix

The "relevance matrix" represents the ground truth and the selected features. The results of the actual measurements are called the ground truth. The relevance matrix also gives an intuitive insight on which operator is more relevant in a given case (given our feature set). To generate this matrix we created 64 different cases (each case is one data point) and for each case we considered the interactions of the six features with each other (2^6). We changed only one parameter for every measurement to be able to detect how ground truth is affected by that particular parameter. Figure 38 shows the relevance matrix.

			Column High				Column Low			
			Row High		Row Low		Row High		Row Low	
			WKLB High	WKLB Low	WKLB High	WKLB Low	WKLB High	WKLB Low	WKLB High	WKLB Low
Skewness High	Join High	UDI High	**HP** | DN	HP | VP | **DN**	**HP** | VP | DN	HP | VP | **DN**	**HP** | DN	**HP** | VP | **DN**	**HP** | VP | DN	**HP** | VP | **DN**
		UDI Low	**HP**	**FR**	**HP**	**HP** | VP	**VP** | DN	**HP**	**HP**	HP | **DN**
	Join Low	UDI High	HP | VP | **DN**	HP | VP | **DN**	HP | **DN**	HP | VP | **DN**	**HP** | VP | DN	**HP** | VP | **DN**	HP | VP | **DN**	HP | VP | **DN**
		UDI Low	**HP**	**FR**	HP | **DN**	**HP** | **VP** | DN	HP | **VP**	**HP** | **VP**	**HP** | **DN**	**HP** | **VP** | **DN**
Skewness Low	Join High	UDI High	HP | **DN**	**HP** | DN	**HP** | DN	HP | **DN**	**HP** | DN	**HP** | **DN**	HP | **DN**	**HP** | DN
		UDI Low	VP | **DN**	**HP** | VP | DN	**HP** | VP	**HP**	**HP** | DN	**HP**	**FR**	**HP** | VP | DN
	Join Low	UDI High	HP | VP | **DN**	HP | **DN**	HP | VP | **DN**	HP | VP | **DN**	**HP** | VP | **DN**	HP | VP | **DN**	**HP** | VP | DN	HP | VP | **DN**
		UDI Low	**FR**	**FR**	**VP**	**VP**	**HP** | **VP** | **DN**	HP | **VP** | **DN**	**HP**	**HP**

Figure 38: The Relevance Matrix

We iteratively changed the features involving the query templates and the schemas of TPC-W to create our new database configurations. In each case, we created different database layouts and measured the total system response time using two threads simultaneously. We turned off the caches of the databases and repeated the measurements until the standard error of the two threads' results was equal or less than 0.05. This process also takes care of warming up the disk caches. The created database layouts

involved FR, HP, VP, and DN operators and their significance testing to each other and to our initial distribution policy. For example, the result for 010110 (skewness low, join heaviness high, UDI/R ratio low, number of columns high, number of rows high, workload balance low) is presented by Table 4.

Table 4: The result of one measurement (010110)

010110	R1[s]	R2[s]	TTEST1	TTEST2	TTEST3	AVERAGE[s]
FR	44.2	44.2	0.0002	0.0471	0.017	44.2
HP	43.7	43.7	0.0006	0.0005	-	43.7
DN	44.1	44.1	0.2307	-	-	44.1
VP	44.1	44.1	-	-	-	44.1

We calculated T-TEST values for significance testing. The first row of the TTEST1 column tests if FR and HP are significantly different from each other. TTEST2 column tests if FR and DN are significantly different from each other and TTEST2 does the same for FR and VP. The second row of TTEST1 computes the same for HP and DN and so on. This measurement reflects that HP, DN, and VP operators are reduced the total system response time and their results are significantly different from Full Replication but DN and VP are not significantly different from each other. We can see that HP-VP and HP-DN are significantly different from each other. Therefore, HP operator minimized the total system response time the most. The Relevance Matrix sixth row and second column shows the result. The matrix reflects all the operators that had significant improvements on the total system response time compared to Full Replication. It also reflects which operator minimized the total system response time the most (underscored operator). If more than one operator is underscored that means they are significantly different from Full Replication but they are not significantly different from each other.

Moreover, both of them lead to an improvement in terms of total system response time. We used the same process to determine each column and row of the Relevance Matrix. In our ground truth there were some instances were multiple outcomes were possible and they were not significantly different from each other. In such cases we considered the operator the one that minimized the total system response time the most - even if it was not significantly different from the other outcomes (HP, VP, or DN), but it was still significantly better than FR. We created the Decision Matrix that includes only a single operator that affected the total system response time the most. Figure 39 shows the Decision Matrix.

			Column High				Column Low			
			Row High		Row Low		Row High		Row Low	
			WKLB High	WKLB Low	WKLB High	WKLB Low	WKLB High	WKLB Low	WKLB High	WKLB Low
Skewness High	Join High	UDI High	HP	DN	HP	DN	HP	HP	HP	HP
		UDI Low	HP	FR	HP	HP	VP	HP	HP	DN
	Join Low	UDI High	DN	DN	DN	DN	HP	DN	DN	DN
		UDI Low	HP	FR	DN	VP	VP	VP	HP	VP
Skewness Low	Join High	UDI High	DN	HP	HP	DN	HP	HP	DN	HP
		UDI Low	DN	HP	HP	HP	HP	HP	FR	HP
	Join Low	UDI High	DN	DN	DN	DN	DN	DN	HP	DN
		UDI Low	FR	FR	VP	VP	VP	VP	HP	HP

Figure 39: The Decision Matrix

6.1.3. Empirical Validation

The Decision Matrix was used to machine learn the rules (Figure 39). This process was done by training and validating a decision tree model on WEKA [84]. Specifically, we used the J.48 decision tree classifier provided in the WEKA library. Cross-validation is done to ensure that our model does not overfit on the training data and generalizes well to

unseen data. Given the small size of data set we performed a leave-one-out cross-validation on the data [85]. Kohavi [85] shows that leave-one-out cross-validation almost always gives an unbiased estimate of the performance of the model learnt. Hence it is quite applicable in our case. This is the same as k-fold cross-validation where the k is equal to the number of data points. In each fold 63 data points are used to train the decision tree and the one point in the validation set is used to test the generalizability of the learned tree. The final results are the average of the 63 folds. The predicted attribute is the "operator" which can take four values (FR, HP, DN, VP). We generated the prediction under leave-one-out cross-validation and compared the actual and the predicted values. One of the cases (see Decision Matrix light green color) that were predicted incorrectly belongs to the group with multiple possible outcomes. In our ground truth there were some instances where multiple outcomes were possible and they were not significantly different from each other. In such cases we considered the label for training to be the one that minimized the total system response time the most - even if it was not significantly different from the other outcomes (HP, VP, or DN), but it was still significantly better than FR. In subsequent training of the model we removed that particular instance from the training data. After removing that instance, we were left with 63 data points. In this case, in each fold 62 data points are used to train the decision tree and the one point in the validation set is used to test the generalization of the learned tree. The final results are the average of the 63 folds. The predicted attribute is the "operator" which can take four values (FR, HP, DN, VP). Appendix F indicates the dataset in WEKA attribute-relation file format (ARFF). Since skewness is a three-valued attribute and our experiments indicated that it was not considered for splitting a node in the

learned decision tree therefore, we removed skewness from the set of features. In other words, our data indicates that skewness might not be a good feature. So, we repeated our experiments without skewness included as a feature and we reported exactly the same results. The tree learnt through our data is shown in Figure 40. The numbers in brackets after the leaf nodes show the total number of instances assigned to that particular node, followed by how many of those instances are incorrectly classified.

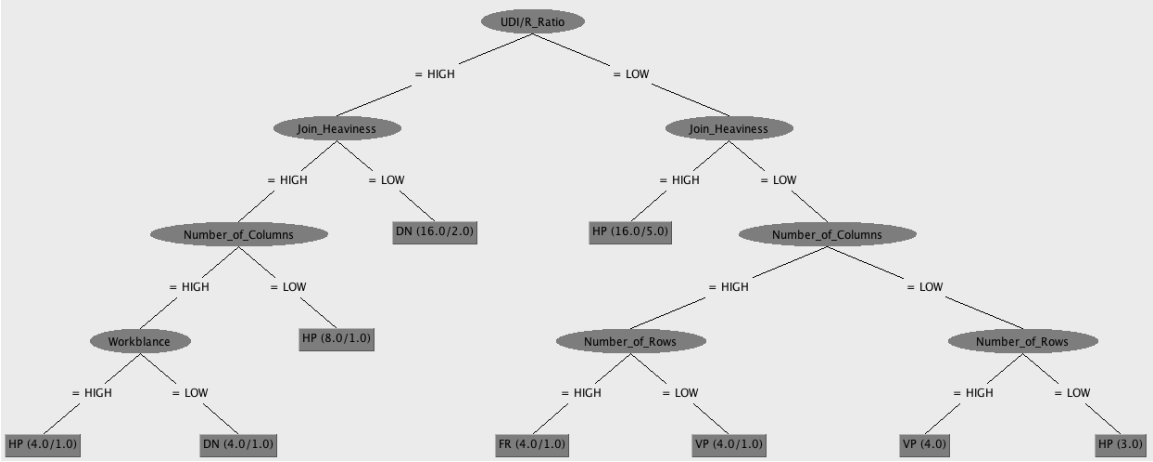


Figure 40: The learned model

This tree can also be interpreted as a set of if-else rules as given by Figure 41.

```
If (UDI/R Ratio='HIGH'):
    If (Join Heaviness='HIGH'):
        If (Number of Columns='HIGH'):
            If (Workload Balance='HIGH') then "HP"
            Else: "DN"
        Else: "HP"
    Else: "DN"
Else:
    If (Join Heaviness='HIGH') then "HP"
    Else:
        If (Number of Columns='HIGH'):
            If (Number of Rows='HIGH') then "FR"
            Else: "VP"
        Else:
            If (Number of Rows='HIGH') then "VP"
            Else: "HP"
```

Figure 41: Interpretation of the decision tree as a set of if-else rules

The decision tree learned above correctly classified 76.1905% of the time. The mean absolute error was 0.1852 and the root mean square error was 0.336. The following confusion matrix was obtained for the leave-one-out cross-validation:

A	B	C	D	←Classified as
3	2	0	0	a = FR
1	24	0	3	b = HP
0	1	7	0	c = VP
0	7	1	14	d = DN

This matrix shows for each class, how instances from that class received classifications. From Full Replication (FR) 3 instances were correctly classified and 2 were put into Horizontal Partitioning class (HP). From "HP" 24 instances were correctly classified by WEKA, 1 was put into "FR" and 3 were assign into "DN" class. From "VP" 7 were correctly classified and 1 was put into "HP". Finally, from "DN" 14 were correctly classified, 1 was put into "VP" and 7 were assign to class "HP". Table 5 shows the result of the prediction under leave-one-out cross-validation and the correctly/incorrectly classified instances.

Table 5: Prediction under leave-one-out cross-validation

Actual	Predicted	Error	Prediction	ID	Fold
FR	FR	-	0.667	26	1
FR	FR	-	0.667	57	2
FR	FR	-	0.667	56	3
FR	HP	+	0.733	10	4
FR	HP	+	0.733	46	5
DN	DN	-	0.867	53	6
DN	HP	+	1	32	7
DN	DN	-	0.867	22	8
DN	DN	-	0.867	48	9

DN	VP	+	1	27	10
DN	HP	+	0.733	40	11
DN	DN	-	0.867	55	12
DN	HP	+	0.733	4	13
DN	DN	-	0.867	19	14
DN	DN	-	0.867	51	15
DN	HP	+	0.733	16	16
DN	HP	+	0.733	2	17
DN	DN	-	0.867	18	18
DN	DN	-	0.867	20	19
DN	DN	-	0.867	49	20
DN	DN	-	0.867	52	21
DN	HP	+	0.733	35	22
DN	HP	+	1	38	23
DN	DN	-	0.867	50	24
DN	DN	-	0.867	24	25
DN	DN	-	0.867	23	26
DN	DN	-	0.867	17	27
HP	HP	-	1	31	28
HP	HP	-	0.667	12	29
HP	HP	-	1	63	30
HP	HP	-	0.857	5	31
HP	HP	-	0.667	44	32
HP	HP	-	0.667	15	33
HP	DN	+	0.933	54	34
HP	HP	-	0.857	39	35
HP	FR	+	1	25	36
HP	DN	+	1	33	37
HP	DN	+	0.933	21	38
HP	HP	-	0.857	36	39
HP	HP	-	0.667	9	40
HP	HP	-	0.857	7	41
HP	HP	-	0.667	41	42
HP	HP	-	0.667	42	43
HP	HP	-	0.857	8	44
HP	HP	-	0.667	14	45
HP	HP	-	0.667	47	46
HP	HP	-	0.667	3	47
HP	HP	-	0.667	43	48
HP	HP	-	0.667	11	49
HP	HP	-	1	62	50
HP	HP	-	0.667	45	51
HP	HP	-	0.857	6	52
HP	HP	-	0.667	34	53
HP	HP	-	0.857	37	54
HP	HP	-	0.667	1	55
VP	VP	-	0.667	58	56
VP	VP	-	1	30	57
VP	VP	-	1	60	58
VP	VP	-	1	29	59
VP	VP	-	0.667	28	60
VP	VP	-	0.667	59	61
VP	HP	+	0.733	13	62
VP	VP	-	1	61	63

Table 5 also includes the number of folds and the IDs of each row that was tested in that fold. Figure 42 shows the result of the class distribution of the operators.

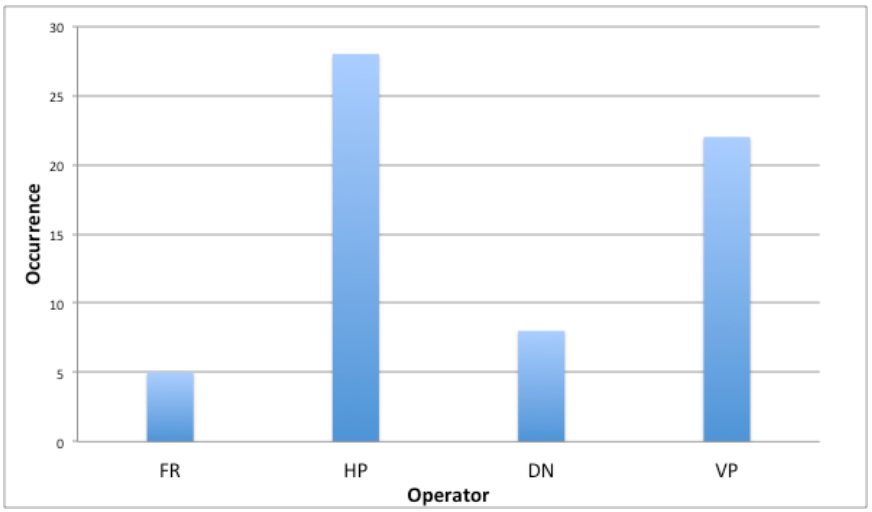


Figure 42: The class distribution of the operators

This distribution reflects that Horizontal Partitioning is the most common operator with 44.444%, Vertical Partitioning is the second most common with 34.92%, Denormalization is the third one with 12.698%, and Full Replication is the last one with 7.936%. The Kappa coefficient produced by WEKA is 0.6307. Kappa value greater than 0 means that our classifier does better than chance. The Receiver operating characteristic curve (ROC) [86] class labels the true positive rate versus the false positive rate for our classifier. The best possible prediction would be a perfect classification: a point in the upper left corner of the ROC space (no false negatives and no false positives). A completely random guess would generate a diagonal line from the left bottom to the top right corner. Figure 43 shows the ROC curves of the operators.

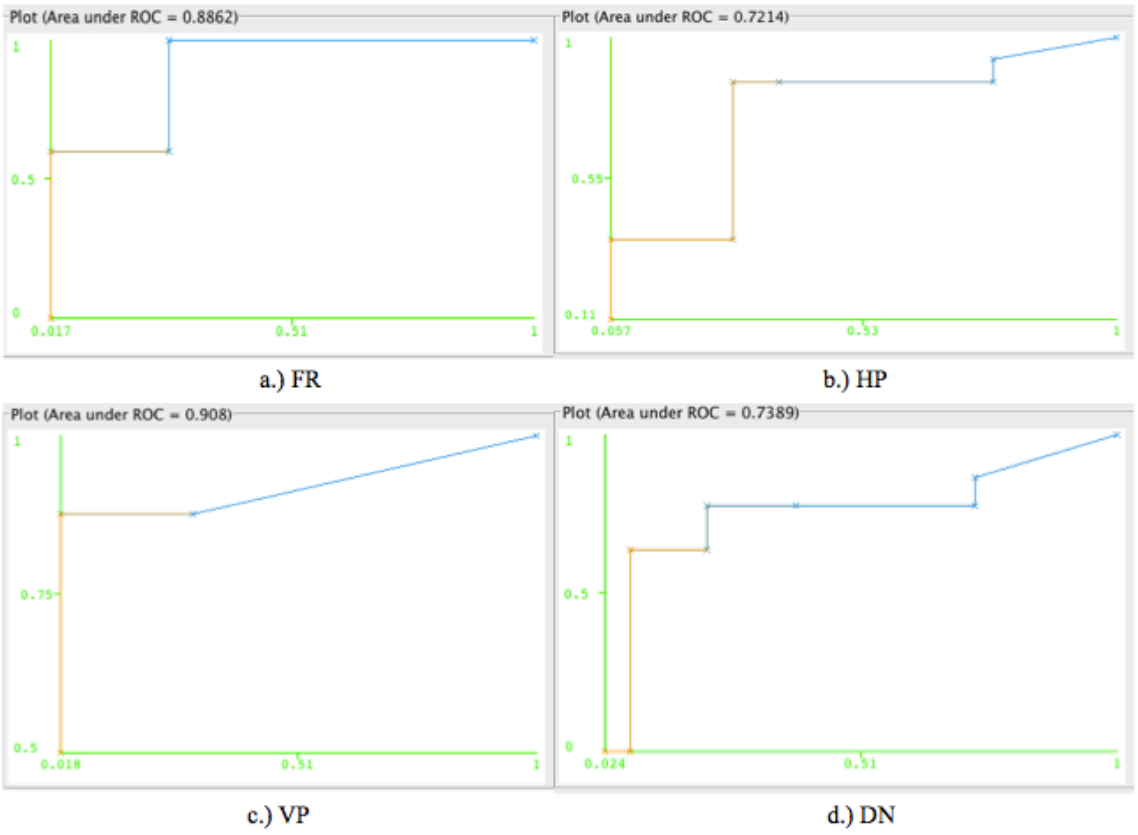


Figure 43: ROC curves of the operators

X axis reflects the False Positive Rate and Y axis shows the True Positive Rate

AUC (Area Under the ROC curve) represents the expected performance in terms of the trade of between the false positive and the true positive rates. Classifiers should perform better than 0.5 and the values greater than 0.5 have better expected performance. Table 6 presents the AUC values for each operators and the weighted average of them.

Table 6: AUC values for each operators and the weighted average

Operator	AUC
FR	**0.886**
HP	**0.721**
VP	**0.908**
DN	**0.739**
Weighted average	**0.764**

6.2. Rules and the Learned Model

The learned model shows us that the most important feature to know is the write or read intensity of a table. The second one is the join heaviness and the third one is the number of columns. At the fourth level the workload balance and the number of rows are present. Based on the tree we can create our database rules:

- "*when the number of update/delete/insert queries on a table is small compared to the number of retrieval queries (e.g selects), then one should consider FR, HP, and VP operators first*";
- "when the number of *update/delete/insert queries on a table is small compared to the number of retrieval queries and the table is heavily involved in joins, then one should horizontally partition";*
- "when the number of *update/delete/insert queries on a table is small compared to the number of retrieval queries and the table has high number of columns but small number of rows, then one should vertically partition";*
- "when the number of *update/delete/insert queries on a table is small compared to the number of retrieval queries and the table has high number of columns and high number or rows, then one should fully replicate";*

The last two rules can show us that vertical partitioning has more benefit compared to full replication if the involved table has only simple retrieval queries, they are focused on a large number of columns, and the table has small number of rows. If the table is read intensive but not join heavy and it has a large number of rows, then one should consider fully replicating the table.

- "when the number of *update/delete/insert queries on a table is small compared to the number of retrieval queries and the table has small number of columns but high number of rows, then one should vertically partition";*
- "when the number of *update/delete/insert queries on a table is small compared to the number of retrieval queries and the table has small number of columns and small number or rows, then one should horizontally partition";*
- "*when the number of update/delete/insert queries on a table is high compared to the number of retrieval queries, then one should consider HP operator first*";

The last two rules can show us that vertical partitioning has more benefit compared to horizontal partitioning if the involved table has only simple retrieval queries, they are focused on a small number of columns, and the table has small number of rows. If the table has large number of rows then one should consider fully replicating the table instead.

When the UDI vs. read ratio is high, we can also create interesting rules:

- "when the number of *update/delete/insert queries on a table is high compared to the number of retrieval queries and the table is join intensive, then one should consider denormalizing the table";*

This rule gives an addition to the previously mentioned recommendation: "*when one frequently joins two tables, then one should denormalize them"*. One should still consider denormalization operator if frequent joins are mixed with frequent UDIs. One logical explanation could be that in the case of a full replication, vertical partitioning, or a denormalization we do have update propagation (distribute new records across all nodes) but the rewrite of the join condition to a simple select can still give enough benefit to make denormalization preferable.

- "when the number of *update/delete/insert queries on a table is high compared to the number of retrieval queries and the table is join intensive with high workload balance, then one should still consider horizontal partitioning";*

This rule shows that even if lot of queries are routed to the same node and the node becomes more loaded compared to the other ones, then we should still consider horizontal partitioning if the table is UDI and join intensive.

6.3. The Modified Data Placement Algorithm

To make the state based search for a new system more efficient over time we can use our model to bias the search for layout for a new database. We can load our saved model (see Figure 44 for the structure of our saved model) and input the unseen data to predict which operators to use. With the predictions, the Data Placement Algorithm became more efficient with laying out database configurations. Database Administrators can get immediate recommendations on partitioning operators that could lead to a performance improvement of their web based application. Also, the model is quite expandable:

additional features and their interactions can be easily added to further increase the number of correctly classified instances. Figure 45 presents the modified version of our Data Placement Algorithm. Step 6.1 integrates the model. In step 6.1 we predict which operator to use on the table that could minimize the total system response time the most. After the prediction we generates the layout configuration and measures the response time of the system. We significantly can reduce the search space because only the predicted states will be created.

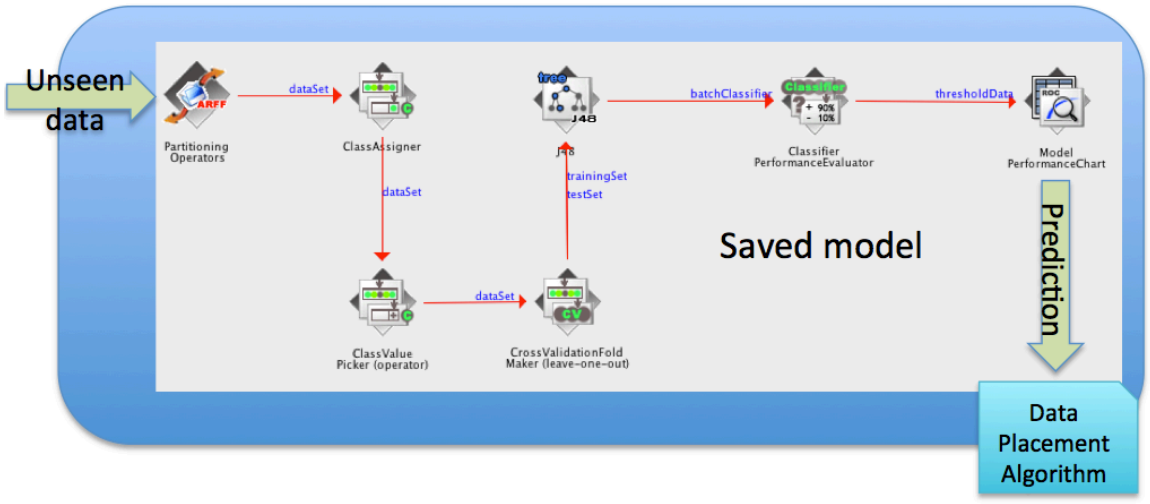


Figure 44: The structure of the model

Step0. Apply Initial Data Distribution Policy: fully replicate all tables across all nodes
Step1. Determine the cost of the workload by running each query template on lightly loaded nodes.
Step2. Initialize an array **dataLayout** that maintains the current data placed on each database server.
Step3. Initialize an array, **currCost** that maintains the current system response time.
The initial cost is set to the result of Step1.
Step4. For each <query template, table> pair, initialize **setOfOptions** to all possible options.
// for instance **setOfOptions** = {replication, horizontal partition, vertical partition, de-normalization}
Step5. For every query in template, remove invalid options from the **setOfOptions**.
Step6. Iterate through the list of query templates and for each query template,
Step 6.1. Predict the best possible placement for every table in the query and conduct the search with the prediction.
Step 6.2. Update the **dataLayout** array to indicate the data on each database server after this placement.
Step 6.3. Update the **currCost** array to indicate the current system response time after this placement.
Step7. Layout the tables across the different database servers according to the **dataLayout** array.

Figure 45: Data Placement Algorithm with prediction

6.4. Virtual Partitioning

Even if the Data Placement Algorithm considers the recommendations we still conduct a search to measure the response time of the system. Also, if we have more than one partitioning key to consider for a table (e.g. we can horizontal partitioning table A based on key K1 or on key K2) then we still have to try both possibilities. To further increase the effectiveness of our algorithm we can get an estimated performance decision on which key to use by connecting our framework with a black-box query optimizer. By connecting to the optimizer we can virtually create a state and get the estimated run times from the optimizer. If the DBA decides not to run the actual search, then the virtual partitioning not only can give recommendations on placement but it can run the search virtually, estimate the states, and generate immediate results.

6.5. Conclusion

This chapter makes one clear contribution, that is of using machine learned rules to help govern the physical design of the database layouts across an arbitrary number of computer nodes. This learning, in turn, allows the database placement algorithm to get better over time as its trains over a set of examples. By utilizing our model the layout algorithm is capable of automatically recommending when it makes sense to apply each of the operators.

The second contribution of this chapter is to select a good set of features based on ideas from database best practices, cut-off rules, experimental evidence, and domain knowledge. Based on the generalization of the learned model, which seems to be good,

we can use the rules themselves to bias a search for a layout for a new database and therefore reduce the search space. Another advantage of such an approach is that the learned model is easily expandable based on new data. In other words, given new data (e.g. more features) new rules can be learned quickly. Finally, the chapter also introduced a methodology to determine the ground truth based on the interactions of the features (Relevance Matrix), determined which operator is significantly different from each other, and concluded this by creating the Decision Matrix for machine learning.

The next chapter describes a comparative analysis of the cut-off rules to be able to assign confidence values for each operator and determine the precedence of each operator.

7. Comparative Analysis

In this part, we first predict operators based on our 14 rules that we determined in chapter 5. Second, we construct an easy representation of the operator precedence as a molecular structure based on the confidence measure of the operators. Finally, we compare our ground truth (Decision Matrix) with our model predictions. This comparison allows us to analyze how cut-off rules based on practices predict the best operators comparing to our ground truth and to the learned model.

7.1. Operator Prediction Based on Cut-off Rules

Based on the selected features (UDI vs. R ratio, join heaviness, number of columns, number of rows, and workload balance, see chapter 6) we determine which of the 14 rules is applicable. Given that we have 5 binary valued features there are 2^5 possibilities as indicated by the Decision Matrix (chapter 6) when there is no skewness. So, we repeat the exercise of determining which of the 14 rules is applicable in each of these cases using the features of the matrix. Table 7 shows this step for two entries. After determining them, we create an Operator Matrix (see Figure 46 for feature set 11000) that reflects how many times one operator was better and worse than the other one. We also construct a pairwise Occurrence Matrix (see Figure 47), which records how many times two operators occur together in the rules. We see that this is a symmetric matrix. We use these two matrices to create a "Voting Matrix" (see Figure 48), in which each element is obtained by dividing the corresponding elements of the Operator Matrix and the pairwise Occurrence Matrix. This matrix gives us a method to compute a value that reflects our

confidence of which operator is better. To determine the confidence value we summarize each row of the Voting Matrix and dividing that by 3 (total number of operators – 1). The numbers represent the precedence order of the operators starting with the best one.

Table 7: Features and rules selection for operator prediction

Features					Rules													
Join heaviness	**UDI vs. R ratio**	**Number of Columns**	**Number of Rows**	**Workload Balance**	HP-VP (#column)	VP-HP (wklb)	FR-HP (wklb)	DN-HP (wklb)	DN-HP (Joinh)	DN-HP (#row)	FR-VP (#col)	FR-VP (UDIR)	DN-FR (Joinh)	DN-VP (#col)	DN-VP (UDIR)	DN-FR (#row)	DN-VP (#row)	**Predicted (rules)**
1	**1**	**0**	**0**	**0**	VP<HP	HP<VP	HP<FR	HP<DN	DN<HP	HP<DN	VP<FR	FR<VP	DN<FR	DN<VP	VP<DN	DN<VP	FR<DN	VP<DN	**HP**
1	**0**	**1**	**1**	**0**	HP<VP	HP<VP	HP<FR	HP<DN	DN<HP	DN<HP	FR<VP	VP<FR	DN<FR	DN<VP	DN<VP	VP<DN	DN<FR	DN<VP	**DN**

For example, the second row and first column of the Operator Matrix (Figure 46) shows that DN operator was 2 times better than VP operator. The first row and second column indicates that VP was 2 times worse than DN based on the rules of table 7 (first entry – VP<DN means VP is better than DN in terms of response time).

		WORSE			
		VP	**DN**	**HP**	**FR**
BETTER	**VP**	X	2	1	1
	DN	2	X	1	1
	HP	1	2	X	1
	FR	1	1	0	X

Figure 46: Operator Matrix (feature set: 11000)

	VP	**DN**	**HP**	**FR**
VP	X	4	2	2
DN	4	X	3	2
HP	2	3	X	1
FR	2	2	1	X

Figure 47: Occurrence Matrix

The Occurrence Matrix (Figure 47) displays that DN-VP and VP-DN rule occurred 4 times overall in the set of cut-off rules. This is a symmetric matrix because VP<DN and DN<VP are the same in terms of rule occurrence.

	VP	DN	HP	FR	SUM	%VotingOnBest	BEST
VP	X	0.5	0.5	0.5	1.5	0.5	HP
DN	0.5	X	0.333333333	0.5	1.333333333	0.444444444	
HP	0.5	0.666666667	X	1	2.166666667	0.722222222	
FR	0.5	0.5	0	X	1	0.333333333	

Figure 48: The Voting Matrix (feature set: 11000)

To create a "Voting Matrix" (see Figure 48) we divide the corresponding elements of the Operator Matrix and the pairwise Occurrence Matrix. For example, let's divide the second row and the first column of the Operator Matrix (2) by the second row and the

first column of the Occurrence Matrix (4). The result is 2/4 = 0.5 as the Voting Matrix second row and first column shows that. We sum all the rows to get the total amount of the earned values then we divide each value by (total number of operators – 1) 3 to get the confidence values. Each confidence value can vary from 0 to 1. In this example horizontal partitioning got the highest value. Therefore, HP is the best operator (see table 6 first entry of the "Predicted" column and %VotingOnBest column of the Voting Matrix). Figure 49 presents the process in an algorithmic format.

algorithm: OperatorPrecedence (Cut-off rules, Feature setting of the Decision Matrix element, Operators)
input: database theory based cut-off rules, feature setting of one of the elements of the Decision Matrix
output: Precedence order of the operators
1. Determine which of the cut-off rules are applicable for the feature setting
2. Create the Operator Matrix to determine how many times one operator is better/worse than the other one
3. Create the pairwise Occurrence Matrix
4. Create the Voting Matrix
5. Compute the confidence factor: summarize each row of the Voting Matrix and divide each sum by (total number of operators – 1)
6. Determine operator precedence: sort the confidence factors in decreasing order
7. Return the operator precedence

Figure 49: Algorithm to determine the operator precedence

7.2. Molecular Structure: The Order of Partitioning

As the result of the process, all operators have a confidence factor. Based on the confidence measure of the operators we can construct an easy representation of the operator precedence as a molecular structure. In the previous example, the precedence order is HP-VP-DN-FR with confidence factors of 0.72, 0.5, 0.44, and 0.33. These values

are based on the previous calculations and they are independent of the inputs or states of the decision tree.

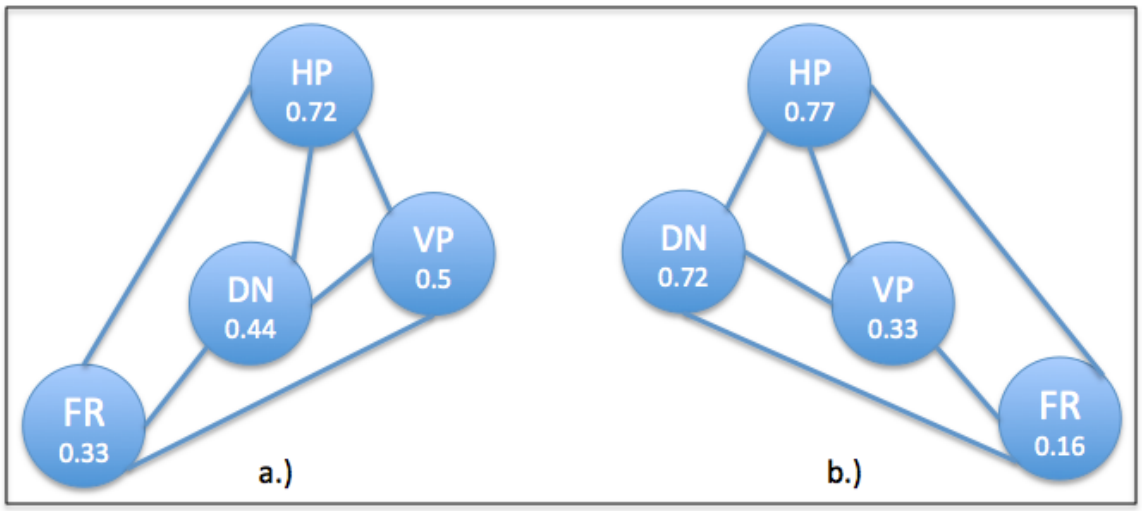


Figure 50: Molecular representation of the operators with confidence values
a.) HP-VP-DN-FR (first entry of table 7)
b.) HP-DN-VP-FR (second entry of table 7)

Figure 50 presents the molecular representation of the operators with confidence values. Each node is an operator and each edge is a path between two operators. The best operator is on the top with the highest confidence value voted to be the most effective one. One can easily check which operator is the second or third best. Despite the confidence values we can go from one operator to the next one following a path that is applicable for us.

7.3. Weighting of the Rules

In the previous sub-chapter we showed that each operator has a confidence value. This confidence value can vary from 0 to 1 but it could happen that two operators have exactly the same values e.g. DN(0.5) and FR(0.5) and the best operator cannot be determined. To

avoid situations with ties, we expand the confidence factor by assigning weights to each rule and use these weights to re-calculate the actual value.

Table 8: The result of one measurement (011111) with Logical Matrix

011111	R1[s]	R2[s]	TTEST1	TTEST2	TTEST3	AVERAGE[s]	FR	HP	DN	VP
FR	99.134	99.128	2.23421E-08	1.15895E-07	1.48618E-05	99.131	X			
HP	74.579	74.582	0.002396276	1.11885E-08	-	74.580	T	X		
DN	74.410	74.425	9.78688E-08	-	-	74.418	T	T	X	
VP	100.109	100.113	-	-	-	100.111	F	F	F	X

As a first step, we add a Logical Matrix to each measurement of the Relevance Matrix (Table 8). A Logical Matrix is a lower triangular matrix. A value of a Logical Matrix is a binary representation: it is True, if a row operator has smaller average value than the column one that we compare to. Otherwise, it is False. For example, if the average value of HP is less than the average value of FR then it is True (first row, first column of the Logical Matrix). If we compare two operators, this representation can quickly inform us which operator minimized the system response time the most. After creating a Logical Matrix for each measurement, then we compare their results with the cut-off rules based on best practices. If two of them agree, then we mark their agreement with 1 otherwise, with 0 (a cut-off rule agreed with the related measurement or not). The weight of a specific rule is calculated by dividing the total number of agreements with the total number of candidates (see Table 9). As the final step, we update our Operator Matrix. We multiply the elements of the matrix by the sum of the related weights. For example, Figure 51 shows the updated Operator Matrix for feature set of 11000 using the weights of Table 9.

Table 9: Calculation of the cut-off rule weights

Join heaviness	UDI vs. R ratio	Number of Columns	Number of Rows	Workload Balance	HP-VP (#column)	VP-HP (wklb)	FR-HP (wklb)	DN-HP (wklb)	DN-HP (Joinh)	DN-HP (#row)	FR-VP (#col)	FR-VP (UDI R)	DN-FR (Joinh)	DN-VP (Joinh)	DN-VP (#col)	DN-VP (UDIR)	DN-FR (#row)	DN-VP (#row)
0	1	1	1	1	HP<VP	VP<HP	FR<HP	DN<HP	HP<DN	DN<HP	FR<VP	FR<VP	FR<DN	VP<DN	VP<DN	DN<VP	DN<FR	DN<VP
Agreement					1	0	0	1	0	1	0	0	0	0	1	1	1	1
1	0	1	1	0	HP<VP	HP<VP	HP<FR	HP<DN	DN<HP	DN<HP	FR<VP	VP<FR	DN<FR	DN<VP	DN<VP	VP<DN	DN<FR	DN<VP
Agreement					1	1	1	1	0	0	0	1	1	0	0	1	1	0
1	1	0	0	0	VP<HP	HP<VP	HP<FR	HP<DN	DN<HP	HP<DN	VP<FR	FR<VP	DN<FR	DN<VP	VP<DN	DN<VP	FR<DN	VP<DN
Agreement					0	1	1	1	1	1	1	1	1	1	0	1	0	0
Weight					2/3	2/3	2/3	3/3	1/3	2/3	1/3	2/3	2/3	1/3	1/3	3/3	2/3	1/3
Percent					66.66%	66.66%	66.66%	100%	33.33%	66.66%	33.33%	66.66%	66.66%	33.33%	33.33%	100%	66.66%	33.33%

		WORSE			
		VP	**DN**	**HP**	**FR**
BETTER	**VP**	X	2(1/3+1/3) 1.33	1(2/3) 0.66	1(1/3) 0.33
	DN	2(1/3+3/3) 2.66	X	1(1/3) 0.33	1(2/3) 0.66
	HP	1(2/3) 0.66	2(3/3+2/3) 3.33	X	1(2/3) 0.66
	FR	1(2/3) 0.66	1(2/3) 0.66	0	X

Figure 51: Operator Matrix example with weights

Table 10 presents the weights for all of the 14 rules based on the calculation steps applied to our measurements. Figure 52 shows the updated Voting Matrix for the same feature set of 110000.

	VP	DN	HP	FR	SUM	%VotingOnBest	BEST
VP	X	0.53	0.22	0.25	1	0.333333333	HP
DN	0.625	X	0.126666667	0.265	1.016666667	0.338888889	
HP	0.25	0.673333333	X	0.53	1.453333333	0.484444444	
FR	0.28	0.33	0	X	0.61	0.203333333	

Figure 52: Updated Voting Matrix (feature set: 110000)

Table 10: Rules and their weights

Rule	Weight [%]
HP-VP (number of columns)	14/32 = 43.75%
VP-HP (workload balance)	16/32 = 50%
FR-HP (workload balance)	17/32 = 53.125%
DN-HP (workload balance)	12/32 = 37.5%
DN-HP (join heaviness)	12/32 = 37.5%
DN-HP (number of rows)	20/32 = 62.5%
FR-VP (number of columns)	16/32 = 50%
FR-VP (UDI vs. read ratio)	18/32 = 56.25%
DN-FR (join Heaviness)	17/32 = 53.125%
DN-VP (join Heaviness)	14/32 = 43.75%
DN-VP (number of columns)	16/32 = 50%
DN-VP (UDI vs. read ratio)	26/32 = 81.25%
DN-FR (number of rows)	21/32 = 65.625%
DN-VP (number of rows)	18/32 = 56.25%

To determine the best operators, we had 2 cases with ties without weights and 0 ties with

weights. Appendix G presents the Ground Truth, cut-off rule predictions with/without weights, and the learned model predictions.

7.4. Comparison of the Predictions

It is interesting to compare the Ground Truth to the predictions of the cut-off rules (with/without weights) and to the learned model. The agreement between the Ground Truth, learned model, and the non-weighted prediction is 22% (Appendix H column E). From the total of 32 predictions all three agreed 7 times. Comparing to the weighted one (Appendix H column F) this value is 37.5%. All three of them agreed 12 times. If we do the comparison based on the Ground Truth and the predictions of the non-weighted cut-off rules (Appendix H column H) we can see that they agreed 9 times (28.1%). However, comparing Ground Truth to the weighted predictions (Appendix H column I) we can conclude that they agreed 15 times (46.8%). This demonstrates that using weights not only can help to select the best operator - in the case of ties - but they can increase the correlation between the cut-off rule based predictions and the Ground Truth. We can also compare the predictions of the learned model with the weighted cut-off rule predictions. They agreed 17 times (53.1%). By introducing and calculating the weights we were able to increase the agreement between the Ground Truth and the cut-off rules by 18.7% (from 28.1% to 46.8%).

7.5. Conclusion

This chapter described a comparative analysis of the cut-off rules to be able to assign confidence values for each operator. With the help of the described operator precedence algorithm, we are able to predict not only the best operators but also the next best choices. This could be useful, especially if the best operator is not applicable for the given application and therefore one should consider the second best operator. It also could happen if the DBA does not want to try out a specific partitioning operator even if it is possible to apply. In this case, we are interested in the second or third best operators. This chapter shows an easy representation of the operator's precedence as a molecular structure based on their confidence values. This representation can be used to compare and represent the order of partitioning. The chapter also compares the ground truth with our model predictions. We introduce two calculations to determine predictions based on the cut-off rules. Predictions without considering weights could create scenarios where we cannot determine the best operators (because of ties). Also, the agreement between the Ground Truth and these predictions is lower. If we consider weighting the rules then we are able to calculate confidence factors that avoid scenarios with ties. Weights also increase the agreement factor between the Ground Truth and the predictions by 18.7%.

The next chapter will show an empirical validation of the learned model on an un-seen test dataset.

8. Empirical validation: ASSISTments, a Free Public Service of Worcester Polytechnic Institute

ASSISTments© [87,88,89] (www.assistments.org) is a Web-based Intelligent Tutoring System that supports thousands of users across Massachusetts. The system is hosted at Worcester Polytechnic Institute. We had full access to the system and its backend. We captured SQL queries during an average school day using the production environment. The size of the log file was 4.3GB and the distribution of the total number (2351381) of SQL queries was: SELECT (1970153), INSERT (48226), UPDATE (305900), DELETE (27102). The system had 83.78% retrieval (select) and 16.22% UDI queries on that day. We collected 194 query templates, not counting the templates of the recently implemented features. Horizontal partitioning (HP) determined 21 possibilities (Table 11), vertical partitioning (VP) created 12 candidates (Table 12), and denormalization (DN) considered 7 possibilities (Table 13) with a 10% denormalization ratio.

Table 11: ASSISTments HP possibilities based on a given workload

Table	Partitioning Key
assistment_ownerships	assistment_id
item_difficulty_logs	problem_id
assistments	Id
assistment_infos	assistment_id
sequence_ownerships	sequence_id
student_classes	Id
Sessions	session_id
teacher_classes	student_class_id
assistment_types	Name
class_files	class_assignment_id
variables	assistment_id
student_class_sections	student_class_id
user_details	user_id
enrollment_states	Name
comments	user_id
taggings	tag_id
tags	Id
problems	assistment_id
item_difficulties	problem_id
tag_categories_tags	tag_id
assignment_logs	assignment_id

Table 12: ASSISTments VP possibilities based on a given workload

Table	Partitioning Key
item_difficulty_logs	difficulty_error, problem_id
enrollment_states	name,description
problem_logs	correct,overlap_time, user_id,assistment_id, first_action, tutor_mode, assignment_type, start_time, first_response_time, actions, original, assignment_id, answer_text, bottom_hint, end_time, hint_count, teacher_comment, tutor_strategy_id, answer_id, problem_id, attempt_count
transfer_models	is_public, subject_id, inferred_from,id
subjects	framework_id,id
images	data
transfer_model_ownerships	content_creator_id, transfer_model_id
item_difficulties	difficulty,problem_id
frameworks	id
class_files	class_assignment_id,id
tags	id,name
assistment_types	name,description

Table 13: ASSISTments DN possibilities based on a given workload

Table	Joining Key
users comments	id,user_id
assistments assistment_infos	id,assistment_id
tag_categories_tags tags	tag_id,id
assistments variables	id,assistment_id
sections section_links	id,child_id
user_details users	user_id,id
taggings tags	tag_id,id

The total number of fully replicated tables was 140. Table 14 shows the set of tables that have one or more partitioning operators applicable based on the previous partitioning options (Table 11, 12, and 13). It also shows the operator predictions using the learned model (Figure 40). The "Feature Set" column shows the High/Low values for the cut-off points (Figure 34, 35, and 36). For example, the Feature Set of "01110" means: UDI/R ratio: Low, Join Heaviness: High, Number of Columns: High, Number of Rows: High, Workload Balance: Low. We manually inputted the cut-off points to our algorithm based on the determined rules (see Chapter 5). The states are compared to State 0 where all tables are fully replicated across the database nodes. The constructed workload had 40,000 queries (combinations of the 194 query templates) shuffled in a random order.

Table 14: ASSISTments test set

ID	Table	Applicable operators	Feature Set	Measured Result	Model Prediction	Agreement
1	item_difficulty_logs	HP/VP/FR	00000	HP	HP	YES
2	assistments	HP/DN/FR	01110	DN	HP	NO
3	assistment_infos	HP/DN/FR	01110	HP	HP	YES
4	assistment_types	HP/VP/FR	00110	FR	FR	YES
5	class_files	HP/VP/FR	01010	HP	HP	YES
6	variables	HP/DN/FR	11110	HP	DN	NO
7	user_details	HP/DN/FR	11110	HP	DN	NO
8	enrollment_sates	HP/VP/FR	00000	HP	HP	YES
9	comments	HP/DN/FR	01110	HP	HP	YES
10	taggings	HP/DN/FR	11000	HP	HP	YES
11	tags	HP/VP/DN/FR	01000	DN	HP	NO
12	item_difficulties	HP/VP/FR	00100	VP	VP	YES
13	tag_categories_tags	HP/DN/FR	01100	HP	HP	YES
14	sequence_ownerships	HP/FR	01100	HP	HP	YES
15	student_classes	HP/FR	11000	HP	HP	YES
16	sessions	HP/FR	10010	HP	DN Not a valid operator	NO
17	teacher_classes	HP/FR	11011	HP	HP	YES
18	student_class_sections	HP/FR	11010	HP	HP	YES
19	problems	HP/FR	11010	HP	HP	YES
20	assignment_logs	HP/FR	11010	HP	HP	YES
21	problem_logs	VP/FR	11110	VP	DN Not a valid operator	NO
22	transfer_models	VP/FR	00100	VP	VP	YES
23	subjects	VP/FR	00100	VP	VP	YES

24	images	VP/FR	10000	VP	DN Not a valid operator	NO
25	transfer_model_ownership	VP/FR	01100	FR	HP Not a valid operator	NO
26	frameworks	VP/FR	01100	VP	HP Not a valid operator	NO
27	sections	DN/FR	11110	FR	DN	NO
28	users	DN/FR	11110	DN	DN	YES
29	section_links	DN/FR	11100	FR	DN	NO
30	assistment_ownership	HP/FR	11110	HP	DN Not a valid operator	NO

For each table, Table 14 shows the applicable operators, the set of features, the Ground Truth (measured result), and the operator predictions based on the learned model. We saved our learned model (Figure 40, Appendix F) using Weka and re-evaluated it on our supplied ASSISTments test set. Appendix I contains the test data in WEKA attribute-relation file format (ARFF) based on the feature sets and the measured results. In each case that we identified a state with significantly not different result from State 0 (FR) but with a lower system response time, we accepted FR as a result. In each case where more than two operators were possible but none of them were significantly different from each other and all of them were significantly different from State 0 we selected the operator with the lowest response time as a result. The model correctly predicted 60% of the time. The results include 6 cases where the operator prediction is not applicable for the particular table (ID 16, 21, 24, 25, 26, and 30 of Table 14). The mean absolute error was 0.237 and the root mean square error was 0.389. The following confusion matrix was obtained for the prediction:

A	B	C	D	←Classified as
1	1	0	2	a = FR
0	13	0	4	b = HP
0	1	3	2	c = VP
0	2	0	1	d = DN

This matrix shows for each class, how instances from that class received classifications. From Full Replication (FR) 1 instance was correctly classified, 1 was put into Horizontal Partitioning (HP) class, and 2 were assigned to the class of Denormalization (DN). From "HP" 13 instances were correctly classified and 4 were assign to "DN" class. From "VP" 3 were correctly classified and 1 was put into "HP" and 2 were assigned to "DN". Finally, from "DN" 1 was correctly classified and 2 were put into "HP". The Kappa coefficient produced by WEKA was 0.55. A Kappa value greater than 0 means that our classifier performs better than chance.

8.1. Conclusion

This chapter applies the learned model and re-evaluates it on an unseen data set to make operator predictions. For this purpose we collected empirical data using ASSISTments, a Web-based Intelligent Tutoring System that supports thousands of users across Massachusetts. After running a one-level search using our database placement algorithm, the system created all possible valid states and measured the total response time. We utilized two simultaneous threads, each with a workload of 40,000 queries and 194 query templates. The system determined 21 states for HP, 12 for VP, and 7 for DN. Including all tables with more than one applicable operator, the total number of data points became

30. By applying our learned model on the constructed test set, we were able to correctly predict the preferred operator 60% of the time. This prediction included 6 cases where the predicted operator was not applicable for the given table. These cases decreased the percentage of the correctly predicted operators. For the sake of curiosity, we removed these 6 cases (24 remaining data points) and reevaluated the model. Using this test set, the system was able to correctly predict the preferred operator 75% of the time.

The predictions that we made can eliminate the need to run a full brute force search and can help to govern the physical design of the database layouts across an arbitrary number of computer nodes. Also, they can help govern the data placement algorithm to consider only states with the highest impact on the total system response time. Furthermore, by utilizing our model, the layout algorithm is not only capable of automatically recommending when it makes sense to apply each of the operators, but it can also select from the group of applicable operators and determine which one should be considered for a particular table.

The next chapter will conclude our work and give ideas for future work.

9. Conclusion of this Dissertation

9.1. Conclusion

This dissertation contributes to the field of database design in computer science. Our methodology, which can help Web-based applications that deal with scalability problems, proposes a solution to resolve database scalability issues and it makes certain assumptions about how to simplify the task.

The techniques described in this dissertation assume that the data is distributed on different database servers in such a manner that any retrieval query is answered by one database server. However, the constraint that any select query is answered by one server is applicable to several applications, especially web applications where all the query templates are known beforehand and the application logic executes the same hard-wired queries over and over again for the same web form request.

This constraint also greatly simplifies query processing and optimization, as no data needs to be exchanged between nodes. Therefore such a system has to only determine which database server needs to execute a query, and then the optimization and execution of the query proceeds on that server as if it was a non-distributed database. To be able to answer each query by a single database node, this dissertation proposes an initial data distribution policy that fully replicates all tables across all database nodes as a starting condition. We compare all measurements and data layouts to this distribution policy.

The dissertation focuses on Web-based applications where the workload consists of a fixed number of query templates. This means that the system is not presented with ad-hoc and unexpected queries. By making our assumption we simplified the processing of individual queries to the databases. Of course, sometimes DBAs want to write queries that can go across all database nodes involving multiple tables, e.g. for analytic purposes, but these analytic queries can be executed as a background task by the DBAs.

This dissertation has investigated the database layout problem and the major contributions are:

- We propose a data placement algorithm that is general. Our placement algorithm considers the given query workload and the time for each query and then determines the best possible placement of data across multiple database nodes using four data operators (full replication, horizontal partitioning, vertical partitioning, denormalization) and our assumptions. What this algorithm calls for is that it learns 1) "What is a good database layout for a particular application given a query workload?" and 2) "Can this algorithm automatically improve itself in making recommendations by using machine learned rules to try to generalize when it makes sense to apply each of these operators?" Our placement algorithm is general so that other techniques for placement can be integrated into our algorithm.

- We propose a search methodology by which we search for better database layouts. By conceptualizing the problem as a state space search problem over database layouts and by doing a full state space search, we can physically create

the layouts and evaluate the overall response time of the system to parameterize the guiding rules of partitioning. There are four operators that we consider that can create a search space of possible database layouts: full replication, horizontal partitioning, vertical partitioning, and denormalization. This dissertation examines the complex interactions between these four different operators to be able to predict which operator is the best to use for a particular database layout. Instead of using best practices to do database layout, we collected empirical data on when these four different operators are effective to determine the ground truth. We wanted to learn rules that are effective at speeding up the entire system, parameterize these rules for cut-off values, and determine the possible sets of important features that we need to take into consideration to learn a general rule. After we created a dataset where these four different operators have been applied to make different databases, we employed machine learning to induce rules to help govern the physical design of the database across an arbitrary number of computer nodes.

- We propose key search algorithms using partially ordered sets and their Hasse diagram representations finding maximal elements of a poset.

- We propose a shared-nothing data replication framework for Web-based applications with state based search and machine learning components to predict when to choose between horizontal partitioning, vertical partitioning, denormalization or full replication layout operators. This established middleware

is general and it can be used by any Web-based application with relational databases.

The proposed framework includes an efficient distributed query router and a tester component. The router directs the queries to different database servers, while ensuring that all the database servers are utilized efficiently. The tester component models the expected usage of an application to make the measurements more realistic to a real word application.

- We propose machine learned rules to help govern the physical design of the database across an arbitrary number of computer nodes. These rules, in turn, allow the placement algorithm to get better over time as it trains over a set of examples.

  We parameterized cut-off points to help the state space search to focus on creating layout configurations that could boost the performance of the application. To determine these points we turned our attention towards database best practices and identified 14 rules. These rules can reduce the size of the search space by eliminating valid table-operator key-pairs because of their possible negative performance effect on the system. The method for parameterizing cut-off rules is quite general and further cut-off point/attribute pairs can be added to the system easily.

  We propose a good set of features based on ideas from database best practices, cut-off rules, experimental evidence, and domain knowledge. Based on the generalization of the learned model we can use the rules themselves to bias a

search for a layout for a new database and therefore reduce the search space. Another advantage of such an approach is that the learned model is easily expandable based on new data.

To test the machine learning idea we gathered data using our framework involving the Industry Standard eBusiness transactional web benchmark's tables and query templates (TPC-W), and generated machine learned rules. Our knowledge is based on the different database configurations and schema attributes of TPC-W that we created. We propose a relevance matrix that represents the ground truth and the selected features. It gives an intuitive insight as to which operator is more relevant in a given case for a specific set of features. We created 64 different cases and for each case we considered the interactions of the identified six features with each other. We captured how ground truth is affected by a particular parameter.

Based on the results of the relevance matrix, we propose the decision matrix that includes only a single operator that affected the total system response time the most. The decision matrix was used to machine learn the rules.

To learn rules we propose to use a decision tree classifier (J.48) provided in the WEKA library. Because of the small set of data points we applied leave-one-out cross-validation on the data set to ensure our model does not overfit on the training data and generalizes well to unseen data.

We propose rules and the decision tree that we learnt through our data that show important features to consider. We showed that our classifier performs better than chance and it correctly classified 76.19% of the time.

- We propose a comparative analysis of the trade-offs to be able to assign confidence values to each operator and to determine their precedences. We also propose an operator precedence algorithm that is able to predict not only the best operators but also the next best choices. The algorithm is especially useful if the best operator is not applicable for the given table and therefore one should consider the second or the third best operator.

  We propose a comparison of the ground truth with our model predictions. We introduce two calculation methods to determine predictions based on the cut-off rules: predictions without considering weights and with considering weights. Predictions without considering weights could create scenarios where we cannot determine the best operators because of a possible tie situation. Introducing weights can increase the agreement factor between the Ground Truth and the predictions by 18.7%.

  We also propose an easy representation of the operator's precedence as a molecular structure based on the confidence values.

- We performed performance evaluation of the system in the workflow section (4.3). We report our experiment using TPC-W with the performed layout search over multiple database nodes and different number of simultaneous emulated browsers (EBs). We used the framework-determined best layout configuration for the measurement with 100 EBs and we minimized the total system response time by 40% compared to state 0. With our methodology we were able to minimize the total system response time significantly.

- We performed empirical validation on ASSISTments, a Free Public Service of Worcester Polytechnic Institute. ASSISTments is a Web-based Intelligent System that is used by thousands of users across Massachusetts. We applied the learned model and re-evaluated it on an un-seen data set to make operator predictions. For this purpose we collected empirical data using ASSISTments. After performing the one-level search using our database placement algorithm, the system created all possible valid states and measured the total response time. We utilized two emulated clients each with a workload of 40,000 queries and 194 query templates. By applying our learned model on the constructed test set we were able to correctly predict the preferred operator 60% of the time. This prediction included six cases where the predicted operator was not applicable for the given table. Re-evaluating our model without these cases the system was able to correctly predict the preferred operator 75% of the time. The predictions can eliminate the need to run a full brute force search and they help govern the data placement algorithm to consider only states with the highest impact on the total system response time. By using our model, the layout algorithm is not only capable of automatically recommending when it makes sense to apply each of the operators but can determine which operator should be considered for a particular table as well.

9.3. Ideas for Future Work

9.3.1. Virtual Partitioning and Black-Box Query Optimizer

To further increase the effectiveness of our algorithm, we can get a quick cost estimate for partitioning keys. For example, if we can horizontally partition table A based on key K1 or on key K2, then our system physically considers both states for partitioning. If we connect our framework with a black-box query optimizer, like IBM DB2, then we can get an estimated performance decision on which key is the best to pick. By connecting the data placement algorithm to a black-box query optimizer, we can virtually create valid states and get the estimated query run times faster from the optimizer. Maybe virtualization could not completely predict the outcome of a real measurement within a distributed environment.

Virtualization can help to expand the applicability of our framework for non-web-based applications where the combinations of valid states are significantly more than in the case of a Web-based application. A brute-force layout search could take much more time to complete.

9.3.2. Combining Operators

Combining different operators with each other could lead to a more advanced data placement algorithm with further performance benefit. For example, we mentioned that we combine vertical partitioning with full replication in section 3.7. We can visualize the combinations of all operators with each other. This means e.g. first we apply the vertical

partitioning operator on table A and then we horizontally partition the vertically partitioned table. Similarly to this, we can do all the possible combinations. It is also possible to re-apply the denormalization operator on an already denormalized table and involve more tables.

9.3.3. Additional Set of Features to Expand the Model

It is important to investigate further significant sets of features to increase the precision of our model. Further table-, query-, and workload-related features can be easily added to our machine-learning environment and they can help to pre-select the applicable operators for a specific table. Further query and workload features are also important to determine more cut-off points. Additional decision trees can be generated based on new sets of features and the precision of the operator predictions can be increased as well.

9.3.4. Ad-hoc and Analytic Queries

We specialized in Web-based applications where the workload consists of a fixed number of query templates. This means the system does not face ad-hoc and unexpected queries. Of course, sometimes DBAs want to write queries that can go across all database nodes involving multiple tables, e.g. for analytical purposes. One possible solution to solve this problem is to maintain a separate database node with fully replicated tables for analytical purposes. These analytic queries can be executed as a background task utilizing the separate database node. To keep the separated database node up-to-date is a challenging task and a possible effective synchronization technique that could be further investigated.

9.3.5. Fault Tolerance

In real systems, we encounter system crashes quite often, and these crashes also need to be handled. In this dissertation, we did not consider fault tolerance. Incorporating fault tolerance into the problem definition could potentially lead to interesting results. For instance, one way of formulating the problem definition with fault tolerance is to impose a constraint that every data item is present in at least two nodes. This is also a promising research direction, worth investigating in future.

Another aspect of fault tolerance is how to handle if an UDI query fails on some nodes, and succeeds in other nodes. How do we detect this scenario, and also how do we remedy such an inconsistency. One can think of a distributed transaction protocol, but such distributed transactions are very heavy weight, and drastically bring down the performance of a system. We therefore need to investigate different semantics as may be applicable for these scenarios, and which can be implemented without drastically impacting the performance of the overall system.

9.3.6. Increased Database Scalability

One potential opportunity for database scalability is to pull some of the database functionality that can be easily replicated out of the database server. For instance, range selection operation that scans a set of rows and selects rows based on a filter condition can be pulled outside the database server. The range selection operation can be easily replicated across multiple servers. However this comes at a cost: the database server may be able to perform the range selection more efficiently, for instance, by building an index,

whereas these options may not be available in the selection operation outside the database server.

We believe that this is a promising direction that we plan to investigate in the future.

9.3.7. Adjustment of Partitioning Decisions

It could happen if there are lots of UDIs such that the database starts to change so much so as to lead to a small table moving to a large table. If the database state changes, one can easily run our method again and include the ability to determine which tables should change their partitioning decisions (potentially with the least cost).

It is my hope that this work will provide some useful insights for solving the layout optimization problem for distributed relational databases using machine learning.

REFERENCES

[1] Tobias Groothuyse, Swaminathan Sivasubramanian, Guilaume Pierre, “GlobeTP: Template-Based Database Replication for Scalable Web Applications”, WWW, Banff, Alberta, Canada, 2007

[2] R. Ramakrishnan and J. Gehrke, “Database Management Systems (Third Edition)”, McGraw-Hill, 2003

[3] Chrisitan Plattner, Gustavo Alonso, and M. Tamer Ozsu, “DBFarm: A Scalable Cluster for Multiple Databases”, Journal of Middleware, 2006, LNCS 4290, pp. 180-200, 2006

[4] Sacca D., and Wiederhold G. Database Partitioning in a Cluster of Processors. ACM TODS,Vol 10,No 1, Mar 1985.

[5] Zhou Wei, Jiang Dejun. Gulllaume Pierre, “Service-Oriented Data Denormalization for Scalable Web Applications”, WWW, Beijing, China, 2008

[6] Agrawal, S., Chaudhuri, S., and Narasayya, V. Automated Selection of Materialized Views and Indexes for SQL Databases. Proceedings of VLDB 2000.

[7] Rao, J., Zhang, C., Lohman, G., and Megiddo, N. Automated Partitioning Design in Parallel Database Systems. Proceedings of the ACM SIGMOD 2002

[8] Zilio, D., Jhingran, A., Padmanabhan, S. Partitioning Key Selection for Shared-Nothing Parallel Database System. IBM Research Report RC 19820. 1994

[9] Sanjay Agrawal, Surajit Chaudhuri, Vivek, Narasayya, et. al., “Database Tuning Advisor for Microsoft SQL Server 2005” (Microsoft), VLDB, Toronto, Canada, 2004, pp. 1110-1121.

[10] Daniel C. Zilio, Jun Rao, Sam Lighstone, Guy Lohman, et. al., “DB2 Design Advisor: Integrated Automatic Physical Database Design”, 30th VLDB Conference, Toronto, Canada, 2004, pp. 1087- 1097.

[11] Chaitanya K. Baru, Gilles Fecteau, et. al., “An Overview of DB2 Parallel Edition”, SIGMOD Conference 1995, pp. 460-462.

[12] Ravishankar Ramamurthy, David J. DeWitt, Qi Su, “A Case for Fractured Mirrors”, 28th VLDB Conference, Hong Kong, China, 2002

[13] Amit Manjhi, Anastassia Ailamaki, Bruce M. Maggs, Todd C. Mowry, Christopher Olston, Anthony Tomasic, "Simultaneous Scalability and Security for Data Intensive Web Applications", SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA

[14] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In Proc. ICDE, 2003.

[15] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang, “Integrating Vertical and Horizontal Paritioning into Automated Physical Database Design", SIGMOD, Paris, France, June 13-18, 2004, pp. 359-370.

[16] Chu, Wesley W., "Optimal File Allocation in a Multicomputer Information System", Information Processing-68, Proc. IFIP Congress, North-Holland, 1968, pp.F80-F85

[17] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. Maggs, and T. Mowry.,"A scalability service for dynamic web applications", In Proc. Conf. on Innovative Data Systems Research, pages 56–69, Asilomar, CA, USA, January 2005.

[18] Reto Krummenacher and Elena Simperl and Karl Baumann, "Towards Large Scale Knowledge Applications", ASWC, Bankgkok, Thailand, 2008

[19] Stratos Papadomanolakis, Anastassia Ailamaki, “AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning”, SSDBM, Santorini Island, Greece, June 21-23, 2004.

[20] C. Plattner and G. Alonso., "Ganymed: Scalable replication for transactional web applications", In Proc. ACM/IFIP/USENIX Intl. Middleware Conf., Toronto, Canada, October 2004.

[21] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar., "Application specific data replication for edge services", In Proc. Intl. WWW Conf., May 2003

[22] S. Sivasubramanian, G. Pierre, and M. van Steen. GlobeDB: Autonomic data replication for web applications. In Proc. Intl. WWW Conf., Chiba, Japan, May 2005

[23] MySQL master-slave replication, http://dev.mysql.com/doc/refman/5.0/en/replication.html

[24] PostgreSQL, http://www.postgresql.org/docs/8.3/static/high-availability.html

[25] Jun Rao, Chun Zhang, Nimrod Megiddo, Guy M. Lohman, "Automating physical database design in a parallel database", SIGMOD Conference 2002: 558-569.

[26] Oracle 11g: http://download.oracle.com/docs/cd/B28359_01/server.111/b28326/repmview.htm

[27] Yuehua Xu, Alan Fern, “Learning Linear Ranking Functions for Beam Search with Application to Planning”, Journal of Machine Learning Research 10 1571-1610, 2009

[28] George Heineman, Gary Pollice, and Stanley Selkow, “Algorithms in a Nutshell”, O'Reilly Media, Inc., ISBN:9780596516246, 2008

[29] Furcy, D., Koenig, S. Limited discrepancy beam search, “, In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pages 125-131, 2005

[30] Yuehua Xu, Alan Fern, “On Learning Linear Ranking Functions for Beam Search”, Proceedings of the 24th International Conference on Machine Learning, Corvallis, OR, 2007.

[31] Rong Zhou and Eric A. Hansen, “Beam-Stack Search: Integrating Backtracking with Beam Search”, 15th International Conference on Automated Planning and Scheduling, Monterey, CA, 2005

[32] J. Gaschnig, “Performance Measurement and Analysis of Certain Search Algorithms”, PhD Thesis, Carnegie-Mellon University, PA., 1979.

[33] R.J. Bayardo and D.P. Miranker, “An optimal backtrack algorithm for tree-structured constraint satisfaction problems”, Artificial Intelligence 71 (1994), pp. 159–181.

[34] http://www.sql.org/sql-database/postgresql/manual/ddl-constraints.html

[35] Razzaq, L., Patvarczki, J., Almeida, S.F., Vartak, M., Feng, M., Heffernan, N.T. and Koedinger, K. (2009). “The ASSISTment builder: Supporting the Life-cycle of ITS Content Creation.”, IEEE Transactions on Learning Technologies Special Issue on Real-World Applications of Intelligent Tutoring Systems. 2(2) 157-166

[36] S. Ceri, M. Negri, and G. Pelagatti, “Horizontal data partitioning in database design,” in Proceedings of the ACM SIGMOD International Conference on Management of Data, 1982, pp. 128– 136.

[37] M.T. Ozsu, and P. Valduriez, “Principles of Distributed Database Systems”, Prentice Hall, ISBN 0-13-659707-6, 1991

[38] L. Bellatreche, K. Karlapalem, and A. Simonet, “Horizontal class partitioning in object-oriented databases”, in proceedings of the 8th International Conference on Database and Expert Systems Applications (DEXA’97), September 1997, pp. 58–67. Lecture Notes in Computer Science 1308.

[39] SQLITE3: http://docs.python.org/library/sqlite3.html

[40] Schkolnick, M., Sorenson, P.: Denormalization: A Performance Oriented Database Design Technique. In: AICA Congress, Bologna, Italy (1980)

[41] Graham R., L. Grotschel and L. Lovasz, “Handbook of combinatorics”, ISBN: 0-262-07170-3, MIT Press, Cambridge, MA, USA, 1995

[42] Jozsef Patvarczki, Neil T. Heffernan, "Automatic Physical Database Tuning Middleware for Web-based Applications", Advances in Databases and Information Systems, 15th International Conference, ADBIS 2011, In Maria Bielikova, Johann Eder, A Min Tjoa (Eds) Advances in Databases and Information Systems Springer-V, pp. 361-374.

[43] http://www.tpc.org/tpcw/

[44] J. Dean, S. Ghemawat, and G. Inc., “Mapreduce: simplified data processing on large clusters”, In OSDI04: Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation, USENIX Association, 2004.

[45] J. Shafer, S. Rixner, and A. L. Cox., “The hadoop distributed filesystem: Balancing portability and performance”, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2010), White Plains, NY, March 2010.

[46] G. S. Davidson, K. W. Boyack, R. A. Zacharski, S. C. Helmreich, and J. R. Cowie, “Data-centric computing with the netezza architecture”, In SANDIA REPORT, Unlimited Release, April., 2006.

[47] Carrie Ballinger, Ron Fryer, “Born To Be Parallel: Why Parallel Origins Give Teradata an Enduring Performance Edge”, IEEE Data Eng. Bull. 20(2): 3-12, 1997.

[48] Surijat Chaudhuri, “Query Optimizers: Time to Rethink the Contract?”, SIGMOD'09, Rhode Island, USA, pp. 961-968, 2009

[49] Ramakrishnan, R., Gehrke, J.: Database Management Systems, 3rd edn. McGraw- Hill, New York (2003)

[50] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. Mapreduce: simplified data processing on large clusters. In OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation, 2004.

[51] Dhruba Borthakur (2008). HDFS Architecture. Apache Software Foundation. http://hadoop.apache.org/common/docs/current/hdfs_design.pdf

[52] Azza Abouzeid, Kamil B. Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. PVLDB, 2(1):922-933, 2009.

[53] Hadoop (2009). Web Page. http://hadoop.apache.org/core/

[54] Ashish Thusoo, Joydeep S. Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wycko_, and Raghotham Murthy. Hive: a warehousing solution over a map reduce framework. Proc. VLDB Endow., 2(2):1626-1629, 2009.

[55] PostgreSQL (2009). Web Page. http://www.postgresql.org/

[56] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data, pages 165-178, New York, NY, USA, 2009. ACM.

[57] Vertica, 2009. The Vertica Analytic Database – Introducing a New Era in DBMS Performance and Efficiency.
http://www.sistina.com/solutions/intelligence/collateral/vertica_new_era_in_dbms_performance.pdf

[58] George S. Davidson, Kevin W. Boyack, Ron A. Zacharski, Stephen C. Helmreich, and Jim R. Cowie. Data-Centric Computing with the Netezza Architecture. SANDIA REPORTSAND2006 3640 Unlimited Release Printed April 2006,
http://www.netezza.com/documents/whitepapers/Sandia_Labs_White_Paper_July_06.pdf

[59] Sue Clarke. Teradata. Butler Group Research Paper. October 2000,
http://www.teradata.com/library/pdf/butler_100101.pdf

[60] Jozsef Patvarczki, Murali Mani, and Neil Heffernan, "Performance Driven Database Design for Scalable Web Applications", Advances in Databases and Information Systems, In J. Grundspenkis, T. Morzy & G. Vossen (Eds) Advances in Databases and Information Systems Springer-Verlag: Berlin. pp 43-58.

[61] IBM Software Information Center,
http://publib.boulder.ibm.com/infocenter/rbhelp/v6r3/index.jsp?topic=/com.ibm.redbrick.doc6.3/wag/wag32.htm

[62] The ASSITments System, Free Public Service of WPI, http://www.assistments.org

[63] Shamkant Navathe, Stefano Ceri, Gio Wierhold, and Jingle Dou, "Vertical Partitioning Algorithms for Database Design", ACM Transactions on Database Systems, Vol. 9, No. 4, December 1984, pages 680-710.

[64] W. T. Joy Mundy, “The Microsoft Data Warehouse Toolkit: With SQL Server 2005 and the Microsoft Business Intelligence Toolset”, John Wiley and Sons, NEW YORK, 2006.

[65] I. Claudia and N. Galemmo, “Mastering Data Warehouse Design -Relational And Dimensional”, John Wiley and Sons, 2003, ISBN: 978-0-471-32421-8.

[66] Schkolnick, M., Sorenson, P.: Denormalization: A Performance Oriented Database Design Technique. In: AICA Congress, Bologna, Italy (1980)

[67] Inmon, W.H.: Information Engineering for the Practitioner: Putting Theory Into Practice. Prentice Hall, Englewood Cliffs (1988)

[68] Westland, J.C.: Economic Incentives for Database Normalization. Information Processing and Management 28(5), 647–662 (1992)

[69] D. Zilio, "*Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems",* PhD thesis, University of Toronto, 1997.

[70] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *Proceedings of ICDE,* pages 826–835, 2007.

[71] N. Bruno and S. Chaudhuri. Automatic Physical Database Tuning: A Relaxation-based Approach. In *Proceedings of SIGMOD,* pages 227– 238, 2005.

[72] N. Bruno and S. Chaudhuri, "Constrained Physical Design Tuning", *PVLDB,* 1(1):4–15, 2008.

[73] I.Alagiannis, D.Dash, K.Schnaitter, A.Ailamaki, and N.Polyzotis. An Automated, Yet Interactive and Portable DB Designer. In *Proceedings of SIGMOD,* pages 1183–1186, 2010.

[74] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *Proceedings of SIGMOD,* pages 359–370, 2004.

[75] S. Agrawal, E. Chu, and V. R. Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. In *Proceedings of SIGMOD,* pages 683–694, 2006.

[76] Daniel *A*. Menascé*, "*TPC-W: A Benchmark for E-Commerce*", IEEE Internet Computing, vol. 6, no. 3, pp. 83-87, May/June, 2002.*

[77] Shavlik J, Mooney R, Towell, G. 1991. Symbolic and neural learning algorithms: an experimental comparison. Machine Learning 6(2): 111–143.

[78] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. Classification and Regression Trees. CRC Press, 1984.

[79] William W. Cohen: Fast Effective Rule Induction. ICML 1995: 115-123

[80] J. Ross Quinlan: Induction of Decision Trees. Machine Learning 1(1): 81-106 (1986)

[81] Jacob Berlin, Amihai Motro, "Database Schema Matching Using Machine Learning with Feature Selection", CAiSE 2002, Toronto, Canada, pp. 452-466.

[82] Wen-Syan Li, Chris Clifton, "Semantic Integration in Heterogeneous Databases Using Neural Networks"**,** VLDB, Proceedings of 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile, 1994, pp. 1-12.

[83] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams."Learning internal representations by error propagation" In: D. E. Rumelhart (editor) & J. L. McClelland (editor). "Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations." MIT Press. ISBN 026268053X, 1986.

[84] Witten, I.H., Frank, E.: Data Mining: Practical machine learning tools and techniques, 2nd edition. Morgan Kaufmann, San Francisco (2005)

[85] Kohavi R. A study of cross-validation and bootstrap for accuracy estimation and model selection. In Proceedings of the Fourteenth International Conference on Artificial Intelligence (IJCAI), pages 1137–1143, San Mateo, CA, 1995. Morgan Kaufmann.

[86] T. Fawcett. ROC graphs: Notes and practical considerations for data mining researchers. Technical report hpl-2003-4, HP Laboratories, Palo Alto, CA, USA, January 2003.

[87] Razzaq, L., Parvarczki, J., Almeida, S.F., Vartak, M., Feng, M., Heffernan, N.T. and Koedinger, K., "The ASSISTment builder: Supporting the Life-cycle of ITS Content Creation", IEEE Transactions on Learning Technologies Special Issue on Real-World Applications of Intelligent Tutoring Systems. 2(2) 157-166.

[88] Patvarczki, J., Politz, J., Heffernan, N.T., "Scalability and Robustness in the Domain of Web Based Tutoring", In Proceedings of the 14th International Conference on Artificial Intelligence in Education, Workshop Proceedings Volume 4, Scalability Issues in AIED. Brighton, UK. 2009, pp. 20-29

[89] Jozsef Patvarczki, Shane Almeida, Joseph E. Beck, Neil T. Heffernan, "Lessons Learned from Scaling Up a Web-Based Intelligent Tutoring System", ITS'08, Montreal, 2008, ISBN 9783540691303, Springer-Verlag, pp. 766-770

APPENDIX A: Case-studies (Join-graphs of Web-based Applications)

1. The ASSISTments System

The ASSISTments system is a web based intelligent tutoring system at WPI CS department.

Location : http://www.assistments.org

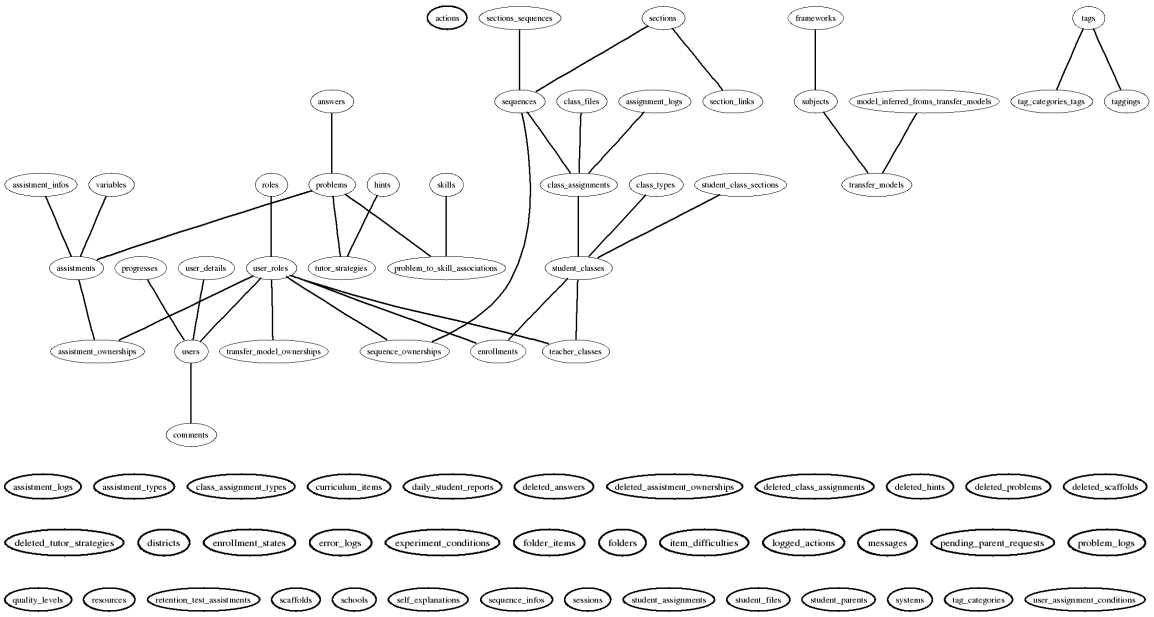
We had access to the system and the backend. We applied ruby code to capture the SQL queries in the logfiles. During a week time interval we collected the production environment log files (20 mongrel ruby processes).

Size of the logfile: 2.107GB

Total number of captured SQL queries: 31515964

A written code part filtered, sorted the queries, and constructed the appropriate format for GraphViz (.dot).

Figure below shows the generated join graph of the ASSISTments system.

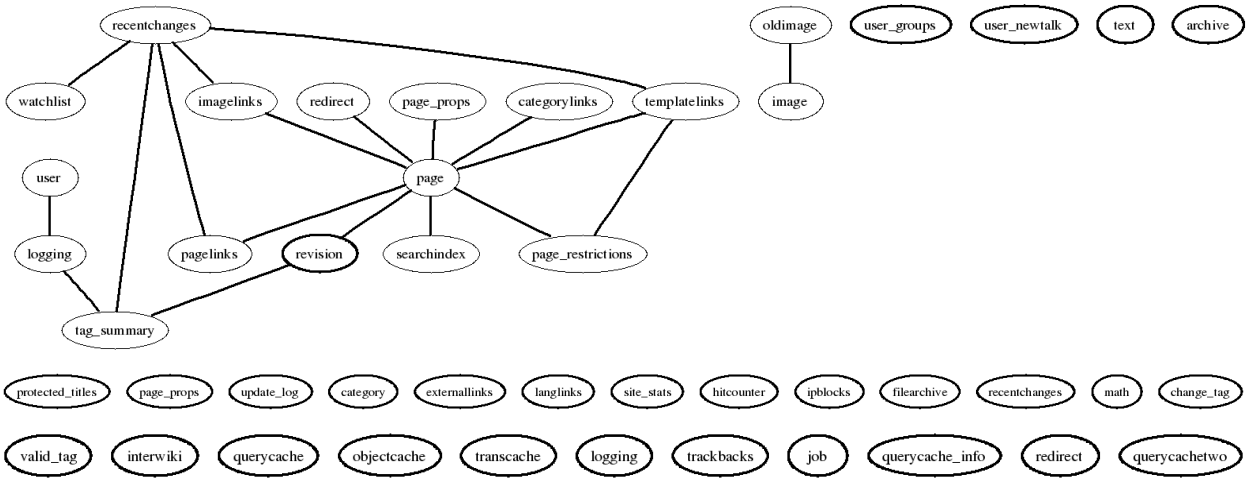


The ASSISTments join graph

2. **MediaWiki**

MediaWiki (http://www.mediawiki.org/wiki/MediaWiki) is a web based wiki software originally for Wikipedia the free encyclopedia.
TeacherWiki is a dynamic place where teachers, students and visitors can interact with each other. Work-study undergraduate students are working on the system as well.

Location: http://teacherwiki.assistment.org
We have access to the system including the backend and the logfiles as well. During a time interval we collected the logfiles and analyzed them.
Size of the logfile: 445M
Total number of captured SQL queries: 119265
Figure below shows the generated join graph of the TeacherWiki system.

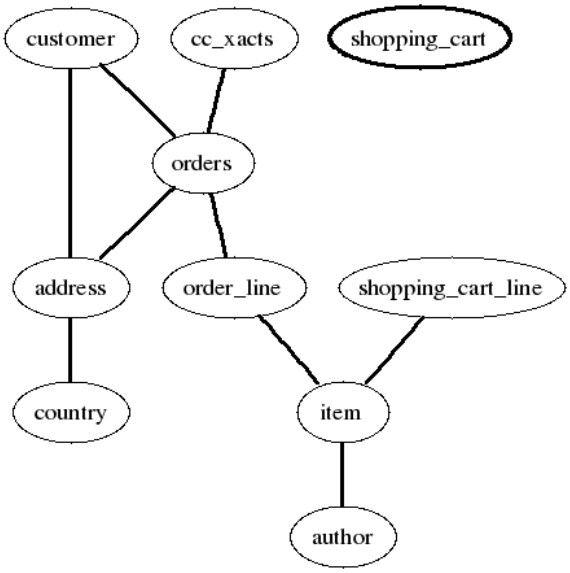


The TeacherWiki Join Graph

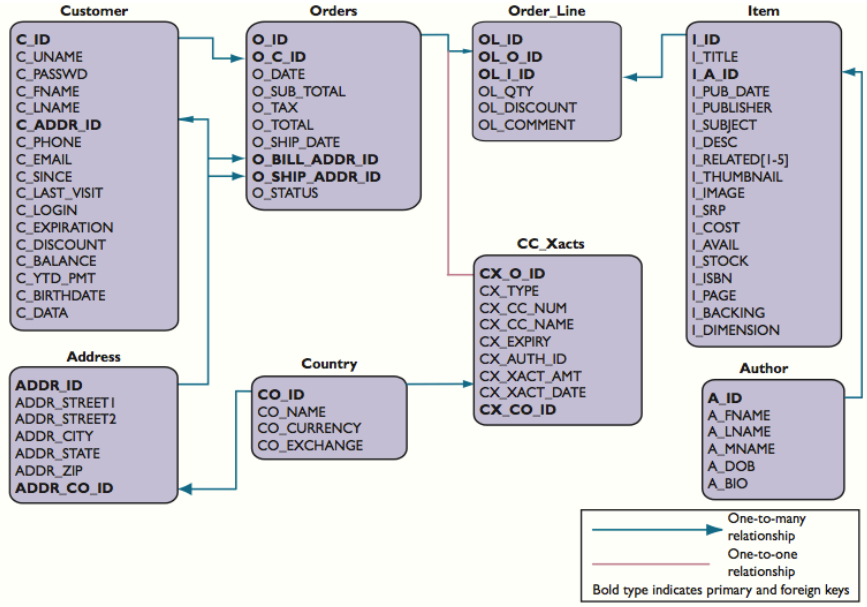
3. TPC-W

TPC-W (http://www.tpc.org/tpcw/) is a transactional web e-Commerce benchmark that models a web based online bookstore.
Used implementation: Java TPC-W Implementation distribution (PHARM University of Wisconsin – Madison) (http://pharm.ece.wisc.edu/tpcw.shtml)

First figure below presents the generated join graph of the TPC-W based on the implementation and the analyzed query templates in the source code. The second figure below shows the ER diagram of TPC-W [76].



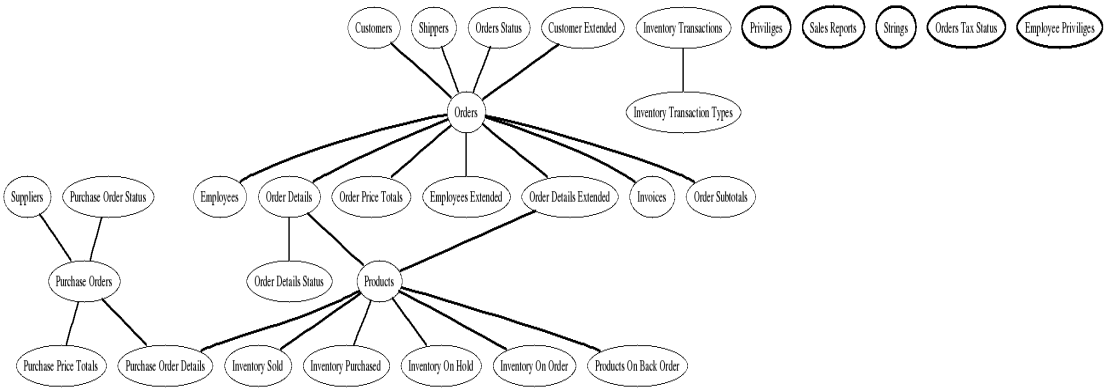
TPC-W Join Graph



The ER diagram of TPC-W [76]

4. NorthWind

NorthWind (http://office.microsoft.com/en-us/templates/TC012289971033.aspx?CategoryID=CT101428651033) is a sample database in Microsoft Access 2007. It contains table sets with given queries (27). Figure below presents the generated join graph based on the analyzed queries and tables structures used Microsoft Access 2007 (Microsoft Office Suite is from WPI with my student license).



NorthWind Join Graph

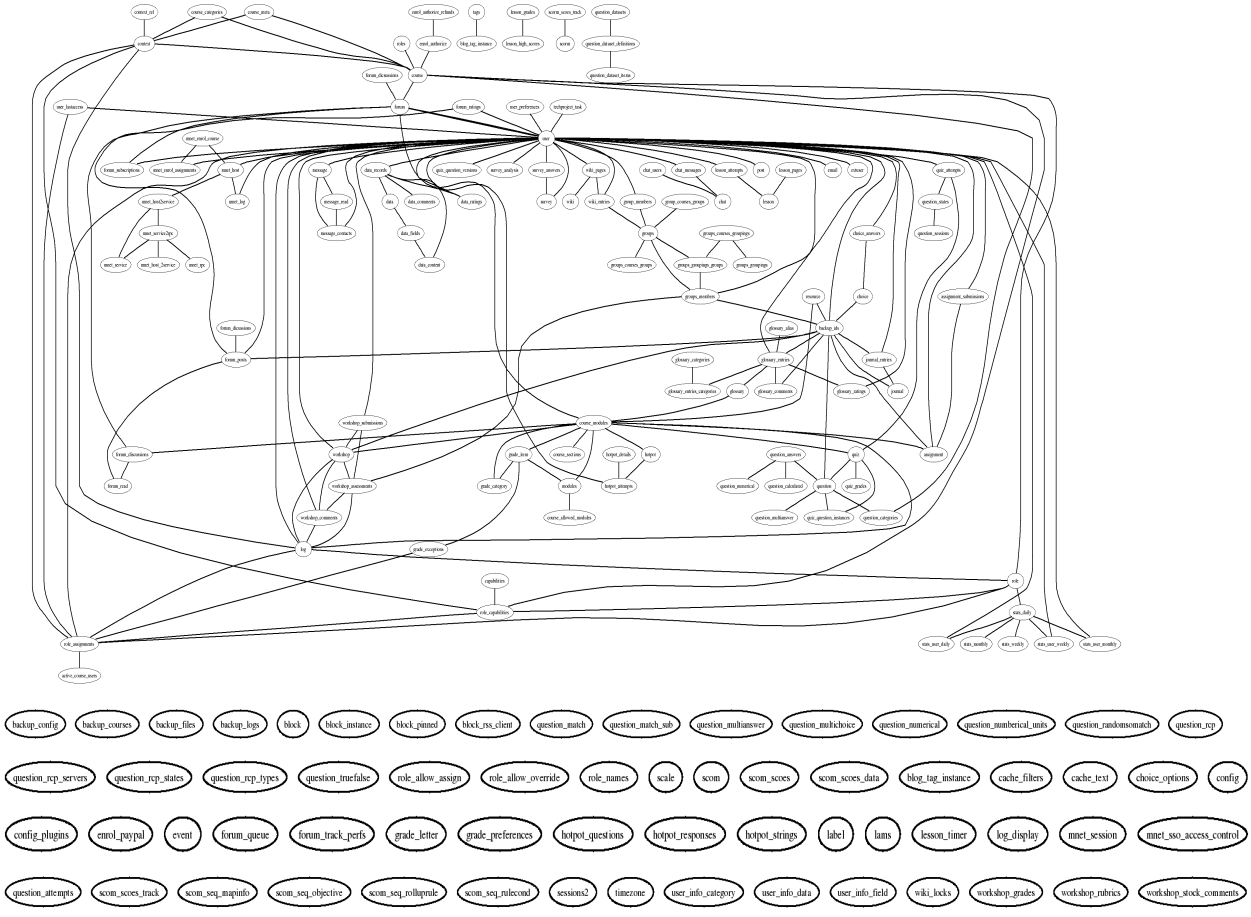
5. Moodle

Moodle 1.8 (http://moodle.org/) is an open-source web based course management system (CMS) for colleges and universities. It is sometimes called Learning Management System (LMS).

By the help of the fabForce DBDesigner4 (http://www.fabforce.net/dbdesigner4/) - that is a visual database design system - we can present the Moodle database table schemas (178 tables). We used the descriptor file posted by Moodle (http://docs.moodle.org/en/Development:Database_Schema)

Total number of 'SELECT SQL' queries in the source code: 793

Figure below presents the join graph of Moodle based on the database schemas and the analyzed query templates in the source code.



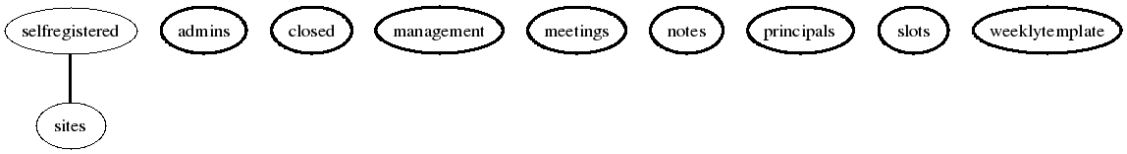
Moodle Join Graph

6. Web-based meeting scheduler by Prof. George Heineman at WPI

Prof. George Heineman (http://www.cs.wpi.edu/~heineman/) has a web based meeting application that he wrote (1999) in CGI-bin perl with a MySQL back-end. Starting around 2005 he began logging all activity for debugging purpose, but the logs haven't been eliminated. It currently contains 86817 log entries. Some of these entries reflect errors, but the majority includes the SQL statements that were executed on the back-end database.

Location: http://users.wpi.edu/~heineman/cgi-bin/meeting/2.0/index.cgi?meetingid=heineman
Size of the logfile: 14MB
Total number of captured SQL queries: 86730

Figure below show the join graph of the application.



Web based meeting application join graph

APPENDIX B: Illustrating Features vs. Related Systems

Feature\System	Microsoft SQL Server 2000 [6]	Microsoft SQL Server 2005 [9]	IBM DB2 Design Advisor [10]	IBM DB2 Design Advisor Enterprise [7]	GlobeTP [1]	Scalability Service [17]	DBProxy [14]
Horizontal Partitioning	N	Y	Y	Y	N	N	N
Vertical Partitioning	N	N	Y	Y	N	N	N
Replication	N	N	N	Y	Y	Y	Y
Denormalization	N	N	N	N	N	N	N
Recommends DB layout	Y	Y	N	Y	N	N	N
Implements DB layout	N	N	N	N	N	N	N
Known query templates	Y	Y	Y	Y	Y	Y	Y
Use Machine Learning Technique	N	N	N	N	N	N	N
Single/Multi node	S	S	S	M	M	M	M

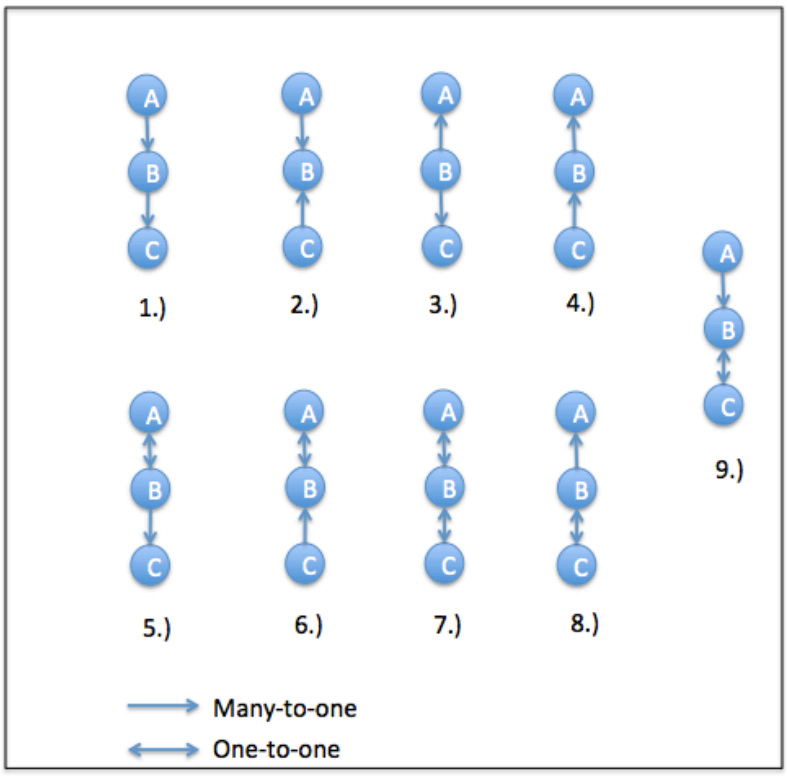
Feature\System	AutoPart [19]	APD [15]	SODD [5]	GanyMed [20]	GlobeDB [22]	Our framework
Horizontal Partitioning	N	Y	Y	Y	N	Y
Vertical Partitioning	N	N	Y	Y	N	Y
Replication	N	N	N	Y	Y	Y
Denormalization	N	N	N	N	N	Y
Recommends DB layout	Y	Y	N	Y	N	Y
Implements DB layout	N	N	N	N	N	Y
Known query templates	Y	Y	Y	Y	Y	Y
Use Machine Learning Technique	N	N	N	N	N	Y
Single/Multi node	S	S	S	M	M	M

APPENDIX C: Query Templates of TPC-W

ID	QUERY TEMPLATE
1	SELECT customer.c_fname, customer.c_lname FROM customer WHERE customer.c_id = ?
2	SELECT * FROM item INNER JOIN author ON item.i_a_id = author.a_id WHERE item.i_id = ?
3	SELECT * FROM customer inner JOIN address ON customer.c_addr_id = address.addr_id inner JOIN country ON address.addr_co_id = country.co_id WHERE customer.c_uname = ?
4	SELECT * FROM item inner JOIN author ON item.i_a_id = author.a_id WHERE item.i_subject = ? ORDER BY item.i_title limit 50
5	SELECT * FROM author inner JOIN item ON item.i_a_id = author.a_id WHERE author.a_lname = '?' limit 50
6	SELECT item.i_id, item.i_title, author.a_fname, author.a_lname FROM item inner JOIN author ON item.i_a_id = author.a_id WHERE item.i_subject = ? ORDER BY item.i_pub_date DESC, item.i_title
7	SELECT item.i_id, item.i_title, author.a_fname, author.a_lname, SUM(order_line.ol_qty) AS orderkey FROM item inner JOIN order_line ON item.i_id = order_line.ol_i_id inner JOIN author ON item.i_a_id = author.a_id WHERE order_line.ol_o_id = ? and item.i_subject = ? GROUP BY order_line.ol_i_id, item.i_id, item.i_title, author.a_fname, author.a_lname ORDER BY orderkey DESC
8	UPDATE item set item.i_cost = ?, item.i_image = ?, item.i_thumbnail = ?, item.i_pub_date = 'XXXX-XX-XX' WHERE item.i_id = ?
9	SELECT order_line.ol_i_id, SUM(order_line.ol_qty) AS orderkey FROM orders inner JOIN order_line ON orders.o_id = order_line.ol_o_id WHERE order_line.ol_i_id = ? and orders.o_c_id = ? GROUP BY order_line.ol_i_id
10	SELECT orders.o_c_id FROM orders inner JOIN order_line ON orders.o_id = order_line.ol_o_id WHERE order_line.ol_i_id = ? and orders.o_id = ?
11	UPDATE item SET item.i_related1 = ?, item.i_related2 = ?, item.i_related3 = ?, item.i_related4 = ?, item.i_related5 = ? WHERE item.i_id = ?
12	SELECT customer.c_uname FROM customer WHERE customer.c_id = X
13	SELECT customer.c_passwd FROM customer WHERE customer.c_uname = ?
14	SELECT item.i_related1 FROM item where item.i_id = ?
15	SELECT orders.o_id FROM customer inner JOIN orders ON customer.c_id = orders.o_c_id WHERE customer.c_uname = ? ORDER BY orders.o_date, orders.o_id DESC
16	SELECT item.i_id FROM item
17	SELECT * FROM orders
18	SELECT * FROM shopping_cart
19	SELECT * FROM customer
20	SELECT * FROM address
21	SELECT COUNT(*) from shopping_cart_line where shopping_cart_line.scl_sc_id = ?
22	UPDATE shopping_cart_line SET shopping_cart_line.scl_qty = ? WHERE shopping_cart_line.scl_sc_id = ?
23	SELECT scl_qty FROM shopping_cart_line WHERE shopping_cart_line.scl_sc_id = ?
24	SELECT address.addr_id FROM address
25	SELECT customer.c_id FROM customer
26	SELECT item.i_stock FROM item WHERE item.i_id = ?
27	INSERT into order_line (ol_id, ol_o_id, ol_i_id, ol_qty, ol_discount, ol_comments) VALUES (?, ?, ?, ?, ?, ?)
28	INSERT into orders (o_id, o_c_id, o_date, o_sub_total, o_tax, o_total, o_ship_type, o_ship_date, o_bill_addr_id, o_ship_addr_id, o_status) VALUES (?, ?, '????-??-??', ?, ?, ?, ?, '????-??-??', ?, ?, '?')
29	INSERT into address (addr_id, addr_street1, addr_street2, addr_city, addr_state, addr_zip, addr_co_id) VALUES (?, ?, ?, ?, ?, ?, ?)

30	SELECT customer.c_addr_id FROM customer WHERE customer.c_id = ?
31	SELECT country.co_id FROM address inner JOIN country ON address.addr_co_id = country.co_id WHERE address.addr_id = ?
32	INSERT into cc_xacts (cx_o_id, cx_type, cx_num, cx_name, cx_expire, cx_xact_amt, cx_xact_date, cx_co_id) VALUES (?, ?, ?, '?', '????-??-??', ?, '????-??-??', ?)
33	DELETE FROM shopping_cart_line WHERE shopping_cart_line.scl_sc_id = ?
34	SELECT country.co_id FROM country WHERE country.co_name = ?
35	SELECT address.addr_id FROM address WHERE address.addr_street1 = ? and address.addr_street2 = ? and address.addr_city = ? and address.addr_state = ? and address.addr_zip = ? and address.addr_co_id = ?
36	SELECT customer.c_discount FROM customer WHERE customer.c_id = ?
37	SELECT * FROM order_line inner JOIN item ON order_line.ol_i_id = item.i_id WHERE order_line.ol_o_id = ?
38	INSERT into shopping_cart (sc_id, sc_time) VALUES (?, '????-??-??')
39	SELECT orders.*, customer.*, cc_xacts.cx_type, address.addr_street1, address.addr_street2, address.addr_state, address.addr_zip, country.co_name, address.addr_street1,address.addr_street2, address.addr_state, address.addr_zip, country.co_name FROM customer, orders, cc_xacts, address, country, address, country inner JOIN orders ON cc_xacts.cx_o_id = orders.o_id inner JOIN customer ON customer.c_id = orders.o_c_id inner JOIN address ON orders.o_bill_addr_id = address.addr_id inner JOIN country ON address.addr_co_id = country.co_id inner JOIN address ON orders.o_ship_addr_id = address.addr_id inner JOIN country ON address.addr_co_id = country.co_id inner JOIN customer ON orders.o_c_id = customer.c_id WHERE orders.o_id = ?
40	SELECT shopping_cart_line.scl_qty FROM shopping_cart_line WHERE shopping_cart_line.scl_sc_id = ? and shopping_cart_line.scl_i_id = ?
41	UPDATE shopping_cart_line SET shopping_cart_line.scl_qty = ? WHERE shopping_cart_line.scl_sc_id = ? and shopping_cart_line.scl_i_id = ?
42	INSERT into shopping_cart_line (scl_sc_id, scl_qty, scl_i_id) VALUES (?,?,?)
43	DELETE FROM shopping_cart_line WHERE shopping_cart_line.scl_sc_id = ? and shopping_cart_line.scl_i_id = ?
44	SELECT * from shopping_cart_line where shopping_cart_line.scl_sc_id = ?
45	UPDATE shopping_cart SET shopping_cart.sc_time = '????-??-??' WHERE shopping_cart.sc_id = ?
46	SELECT * FROM shopping_cart_line inner JOIN item ON shopping_cart_line.scl_i_id = item.i_id WHERE shopping_cart_line.scl_sc_id = ?
47	UPDATE customer SET customer.c_login = 'joe', customer.c_expiration = ? WHERE customer.c_id = ?
48	INSERT into customer (c_id, c_uname, c_passwd, c_fname, c_lname, c_addr_id, c_phone, c_email, c_since, c_last_login, c_login, c_expiration, c_discount, c_balance, c_ytd_pmt, c_birthdate, c_data) VALUES (?, '?', '?', '?', '?', ?, ?, '?', '????-??-??' , '????-??-??' , '????-??-?? ??:??:??.???', '????-??-?? ??:??:??.???', ?, ?, ?, '????-??-??', '????-??-??')

APPENDIX D: Relationship Possibilities



1.) Table B: Foreign KEY on ROLES(ID)
Table C: Foreign KEY on USERS(ID)
HP on ROLES(ID) : (A.ID)

2.) Table B: Foreign KEY on ROLES(ID) and Foreign KEY on LOGS(LOGS_ID)
BROKEN unless creating a relationship between e.g. Table A and C

3.) Table A: Foreign KEY on USERS(ID)
Table C: Foreign KEY on USERS(ID)
HP on USERS(ID): (B.ID)

4.) Table A: Foreign KEY on USERS(ID)
Table B: Foreign KEY on LOGS(LOGS_ID)
HP on LOGS(LOGS_ID) : (C.LOGS_ID)

5.) Table C: Foreign KEY on LOGS(LOGS_ID)
HP on ROLES(ID) or on USERS(ID) : (A.ID) or (B.ID) based on the WHERE clause

6.) Table B: Foreign KEY on LOGS(LOGS_ID)
BROKEN unless creating a relationship between e.g. Table C and A

7.) Select A.ID or B.ID or C.LOGS_ID for HP based on the WHERE clause

8.) Table A: Foreign KEY on USERS(ID)
HP on LOGS(LOGS_ID) or on USERS(ID): (C.LOGS_ID) or (B.ID) based on the WHERE clause

9.) Table B: Foreign KEY on ROLES(ID)
BROKEN

APPENDIX E: The Constraints File of TPC-W

customer.c_id=orders.o_c_id|one2many
address.addr_id=customer.c_addr_id|one2many
address.addr_id=orders.o_bill_addr_id|one2many
address.addr_id=orders.o_ship_addr_id|one2many
country.co_id=cc_xacts.cx_co_id|one2many
country.co_id=address.addr_co_id|one2many
orders.o_id=order_line.ol_o_id|one2many
cc_xacts.cx_o_id=order_lines.ol_o_id|many2many
item.i_id=order_line.ol_i_id|one2many
author.a_id=item.i_a_id|one2many
order_line.ol_o_id=item.i_id|many2one
order_line.ol_o_id=order_line.ol_i_id|many2many
item.i_subject=order_line.ol_i_id|many2many
order_line.ol_o_id=item.i_a_id|many2many
order_lin*e*.ol_o_id=author.a_id|many2many
item.i_subject=author.a_id|many2many
order_line.ol_i_id=orders.o_id|many2many
orders.o_c_id=order_line.ol_o_id|many2many
customer.c_uname=customer.c_id|many2many
customer.c_uname=orders.o_c_id|many2many
orders.o_id=cc_xacts.cx_o_id|one2one
orders.o_id=orders.o_id|one2one
orders.o_c_id=customer.c_id|many2one
orders.o_id=orders.o_c_id|many2many
orders.o_id=orders.o_bill_addr_id|many2many
orders.o_id=bill.addr_id|many2many
orders.o_id=bill.addr_co_id|many2many
orders.o_id=bill_co.co_id|many2many
orders.o_id=orders.o_ship_addr_id|many2many
orders.o_id=ship.addr_id|many2many
orders.o_id=ship.addr_co_id|many2many
orders.o_id=ship_co.co_id|many2many
shopping_cart_line.scl_sc_id=shopping_cart_line.scl_i_id|many2many
shopping_cart_line.scl_sc_id=item.i_id|many2many
address.addr_id=address.addr_co_id|many2many
address.addr_id=country.co_id|many2many
item.i_a_id=author.a_id|many2one
item.i_id=item.i_a_id|many2many
item.i_id=author.a_id|many2many
item.i_a_id=item.i_a_id|one2one
customer.c_uname=customer.c_addr_id|many2many
customer.c_uname=address.addr_id|many2many
customer.c_uname=address.addr_co_id|many2many
customer.c_uname=country.co_id|many2many

item.i_subject=item.i_a_id|many2many
author.a_lname=item.i_a_id|many2many
author.a_lname=author.a_id|many2many
item.i_id=item.i_id|one2one
item.name=author.a_id|many2many
shopping_cart_line.scl_i_id=item.i_id|many2one
address.addr_street1=address.addr_street2|many2many
address.addr_co_id=country.co_id|one2one
country.co_id=country.co_id|one2one
orders.o_id=customer.c_id|many2many
orders.o_id=address.addr_id|many2many
orders.o_id=address.addr_co_id|many2many
orders.o_id=country.co_id|many2many
orders.o_id=ord.o_ship_addr_id|many2many
orders.o_id=cust.c_id|many2many
orders.o_c_id=orders.o_c_id|one2one
customer.c_id=customer.c_id|one2one
order_line.ol_i_id=item.i_id|many2one

APPENDIX F: Attribute-Relation File Format (ARFF)

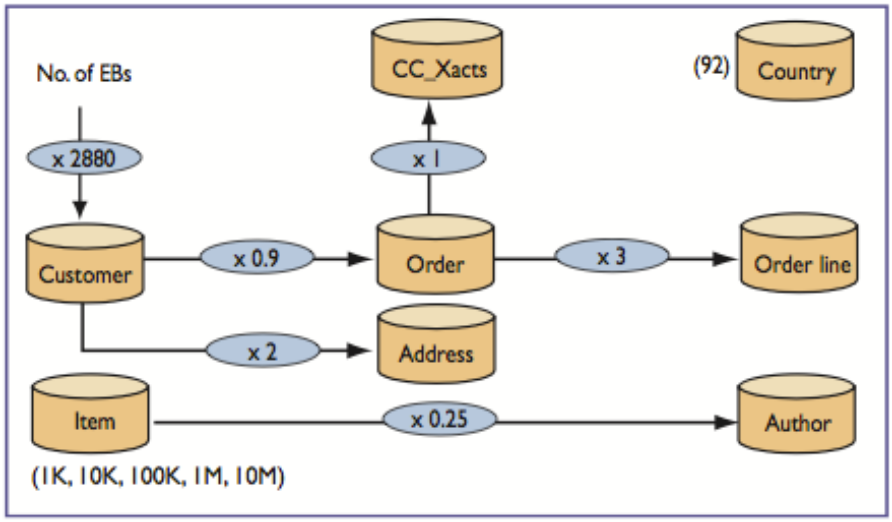
@relation Operator-weka.filters.unsupervised.attribute.AddID-Cfirst-NID

@attribute ID numeric
@attribute Join_Heaviness {HIGH, LOW}
@attribute UDI/R_Ratio {HIGH, LOW}
@attribute Number_of_Columns {HIGH, LOW}
@attribute Number_of_Rows {HIGH, LOW}
@attribute Workblance {HIGH, LOW}
@attribute Operator {FR, HP, VP, DN}

@data
1,HIGH,HIGH,HIGH,HIGH,HIGH,HP
2,HIGH,HIGH,HIGH,HIGH,LOW,DN
3,HIGH,HIGH,HIGH,LOW,HIGH,HP
4,HIGH,HIGH,HIGH,LOW,LOW,DN
5,HIGH,HIGH,LOW,HIGH,HIGH,HP
6,HIGH,HIGH,LOW,HIGH,LOW,HP
7,HIGH,HIGH,LOW,LOW,HIGH,HP
8,HIGH,HIGH,LOW,LOW,LOW,HP
9,HIGH,LOW,HIGH,HIGH,HIGH,HP
10,HIGH,LOW,HIGH,HIGH,LOW,FR
11,HIGH,LOW,HIGH,LOW,HIGH,HP
12,HIGH,LOW,HIGH,LOW,LOW,HP
13,HIGH,LOW,LOW,HIGH,HIGH,VP
14,HIGH,LOW,LOW,HIGH,LOW,HP
15,HIGH,LOW,LOW,LOW,HIGH,HP
16,HIGH,LOW,LOW,LOW,LOW,DN
17,LOW,HIGH,HIGH,HIGH,HIGH,DN
18,LOW,HIGH,HIGH,HIGH,LOW,DN
19,LOW,HIGH,HIGH,LOW,HIGH,DN
20,LOW,HIGH,HIGH,LOW,LOW,DN
21,LOW,HIGH,LOW,HIGH,HIGH,HP
22,LOW,HIGH,LOW,HIGH,LOW,DN
23,LOW,HIGH,LOW,LOW,HIGH,DN
24,LOW,HIGH,LOW,LOW,LOW,DN
25,LOW,LOW,HIGH,HIGH,HIGH,HP
26,LOW,LOW,HIGH,HIGH,LOW,FR
27,LOW,LOW,HIGH,LOW,HIGH,DN
28,LOW,LOW,HIGH,LOW,LOW,VP
29,LOW,LOW,LOW,HIGH,HIGH,VP
30,LOW,LOW,LOW,HIGH,LOW,VP
31,LOW,LOW,LOW,LOW,HIGH,HP
32,HIGH,HIGH,HIGH,HIGH,HIGH,DN
33,HIGH,HIGH,HIGH,HIGH,LOW,HP
34,HIGH,HIGH,HIGH,LOW,HIGH,HP
35,HIGH,HIGH,HIGH,LOW,LOW,DN
36,HIGH,HIGH,LOW,HIGH,HIGH,HP
37,HIGH,HIGH,LOW,HIGH,LOW,HP
38,HIGH,HIGH,LOW,LOW,HIGH,DN
39,HIGH,HIGH,LOW,LOW,LOW,HP
40,HIGH,LOW,HIGH,HIGH,HIGH,DN
41,HIGH,LOW,HIGH,HIGH,LOW,HP

42,HIGH,LOW,HIGH,LOW,HIGH,HP
43,HIGH,LOW,HIGH,LOW,LOW,HP
44,HIGH,LOW,LOW,HIGH,HIGH,HP
45,HIGH,LOW,LOW,HIGH,LOW,HP
46,HIGH,LOW,LOW,LOW,HIGH,FR
47,HIGH,LOW,LOW,LOW,LOW,HP
48,LOW,HIGH,HIGH,HIGH,HIGH,DN
49,LOW,HIGH,HIGH,HIGH,LOW,DN
50,LOW,HIGH,HIGH,LOW,HIGH,DN
51,LOW,HIGH,HIGH,LOW,LOW,DN
52,LOW,HIGH,LOW,HIGH,HIGH,DN
53,LOW,HIGH,LOW,HIGH,LOW,DN
54,LOW,HIGH,LOW,LOW,HIGH,HP
55,LOW,HIGH,LOW,LOW,LOW,DN
56,LOW,LOW,HIGH,HIGH,HIGH,FR
57,LOW,LOW,HIGH,HIGH,LOW,FR
58,LOW,LOW,HIGH,LOW,HIGH,VP
59,LOW,LOW,HIGH,LOW,LOW,VP
60,LOW,LOW,LOW,HIGH,HIGH,VP
61,LOW,LOW,LOW,HIGH,LOW,VP
62,LOW,LOW,LOW,LOW,HIGH,HP
63,LOW,LOW,LOW,LOW,LOW,HP

APPENDIX F: The Cardinality of the Various Database Tables (TPC-W)



The cardinality of the tables and Emulated Browsers [76]

APPENDIX G: Predictions

GroundTruth	PredictedFromLearnedModel	PredictedFromRules(Non-Weighted)	PredictedFromRules(Weighted)
A	B	C	D
DN	HP	DN	DN
HP	DN	DN	DN
HP	DN	FR	DN
DN	HP	HP	HP
HP	DN	DN	DN
HP	HP	DN	HP
DN	HP	VP/FR	VP
HP	HP	HP	HP
DN	HP	DN	DN
HP	HP	HP	HP
HP	HP	FR	VP
HP	HP	HP	HP
HP	HP	VP	VP
HP	HP	VP	VP
FR	HP	VP	VP
HP	HP	VP	VP
DN	DN	FR	DN
DN	DN	HP	DN
DN	DN	FR	FR
DN	DN	HP	HP
DN	DN	DN	DN
DN	DN	HP	DN
HP	DN	FR	FR
DN	DN	HP	HP
FR	FR	FR	FR
FR	FR	HP	HP
VP	HP	FR	VP
VP	HP	HP	HP
VP	VP	VP	VP
VP	VP	VP	VP
HP	HP	VP	VP
HP	HP	VP/HP	HP

APPENDIX H: Comparison of the Predictions

E	F	G	H	I
NO	NO	NO	YES	YES
NO	NO	NO	NO	NO
NO	NO	NO	NO	NO
NO	NO	NO	NO	NO
NO	NO	NO	NO	NO
NO	YES	NO	NO	YES
NO	NO	NO	NO	NO
YES	YES	YES	YES	YES
NO	NO	NO	YES	YES
YES	YES	YES	YES	YES
NO	NO	NO	NO	NO
YES	YES	YES	YES	YES
NO	NO	NO	NO	NO
NO	NO	NO	NO	NO
NO	NO	NO	NO	NO
NO	NO	NO	NO	NO
NO	YES	NO	NO	YES
NO	YES	NO	NO	YES
NO	NO	NO	NO	NO
NO	NO	NO	NO	NO
YES	YES	YES	YES	YES
NO	YES	NO	NO	YES
NO	NO	NO	NO	NO
NO	NO	NO	NO	NO
YES	YES	YES	YES	YES
NO	NO	NO	NO	NO
NO	NO	NO	NO	YES
NO	NO	NO	NO	NO
YES	YES	YES	YES	YES
YES	YES	YES	YES	YES
NO	NO	NO	NO	NO
NO	YES	NO	NO	YES
Percentage of agreement [%]	**21.875**	**37.5**	**18.75**	**28.125**	**46.875**

COLUMN LEGEND: E: IsThereAgreementOnBest(Non-Weighted)? -, F: IsThereAgreementOnBest(Weighted)?, G: IsThereAgreementOnBest(Overall)?, H: DoesBestPracticeAgreeWithGroundTruth(Non-Weighted)?, I: DoesBestPracticeAgreeWithGroundTruth(Weighted)?

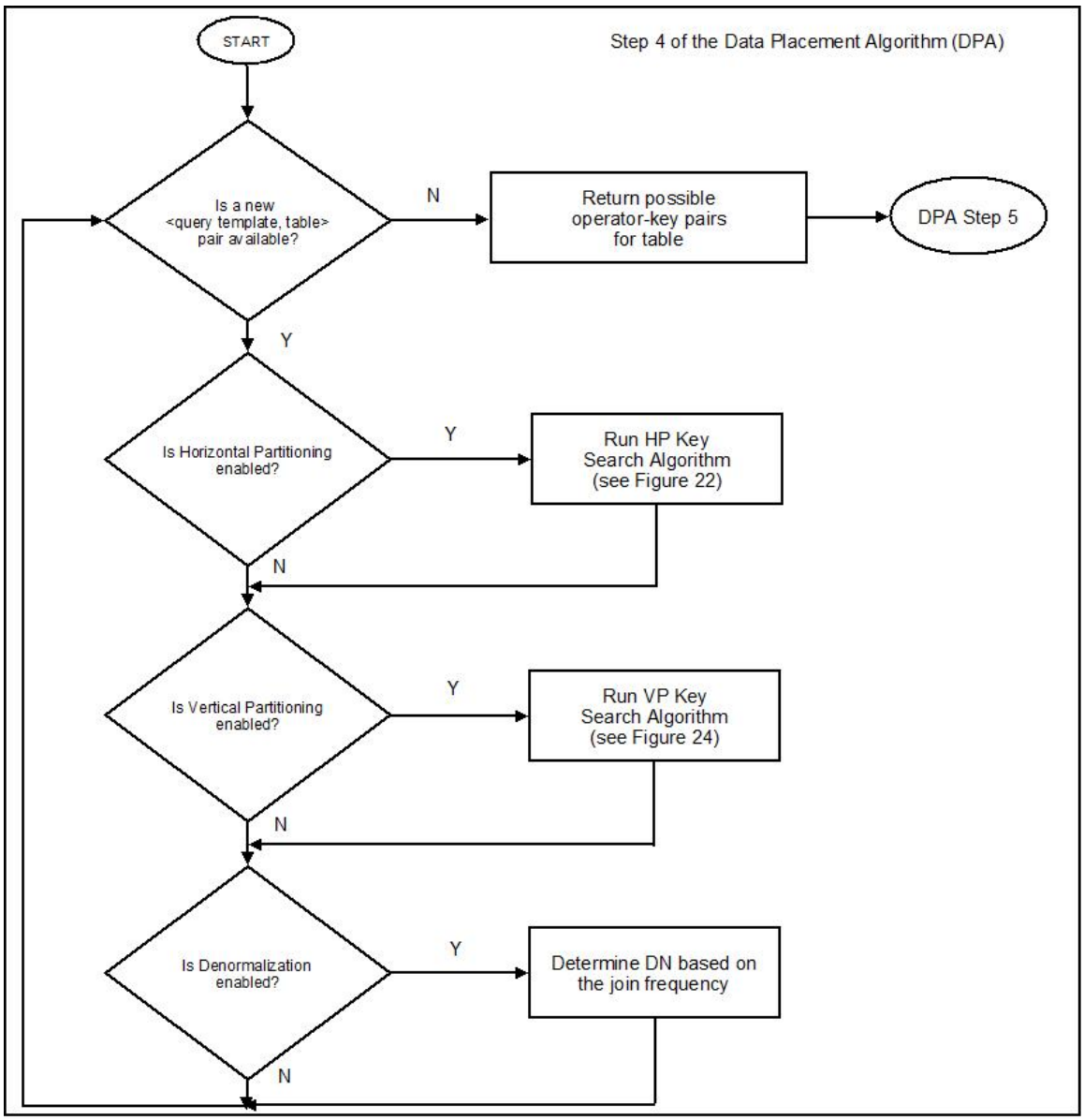
APPENDIX I: ARFF of ASSISTments test data

@relation Operator-weka.filters.unsupervised.attribute.AddID-Cfirst-NID

@attribute ID numeric
@attribute Join_Heaviness {HIGH, LOW}
@attribute UDI/R_Ratio {HIGH, LOW}
@attribute Number_of_Columns {HIGH, LOW}
@attribute Number_of_Rows {HIGH, LOW}
@attribute Workblance {HIGH, LOW}
@attribute Operator {FR, HP, VP, DN}

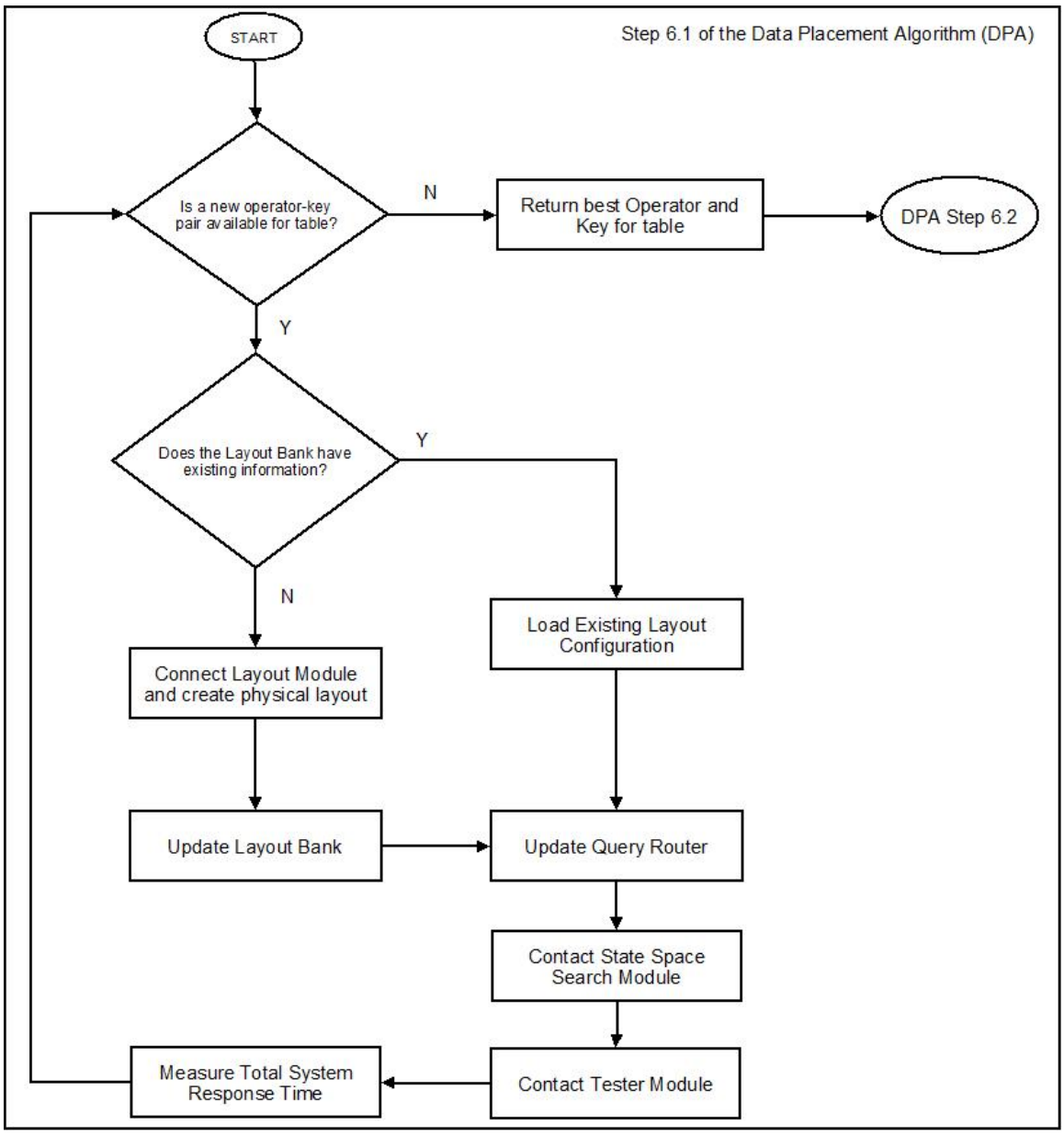
@data
1,LOW,LOW,LOW,LOW,LOW,HP
2,HIGH,LOW,HIGH,HIGH,LOW,DN
3,HIGH,LOW,HIGH,HIGH,LOW,HP
4,LOW,LOW,HIGH,HIGH,LOW,FR
5,HIGH,LOW,LOW,HIGH,LOW,HP
6,HIGH,HIGH,HIGH,HIGH,LOW,HP
7,HIGH,HIGH,HIGH,HIGH,LOW,HP
8,LOW,LOW,LOW,LOW,LOW,HP
9,HIGH,LOW,HIGH,HIGH,LOW,HP
10,HIGH,HIGH,LOW,LOW,LOW,HP
11,HIGH,LOW,LOW,LOW,LOW,DN
12,LOW,LOW,HIGH,LOW,LOW,VP
13,HIGH,LOW,HIGH,LOW,LOW,HP
14,HIGH,LOW,HIGH,LOW,LOW,HP
15,HIGH,HIGH,LOW,LOW,LOW,HP
16,LOW,HIGH,LOW,HIGH,LOW,HP
17,HIGH,HIGH,LOW,HIGH,HIGH,HP
18,HIGH,HIGH,LOW,HIGH,LOW,HP
19,HIGH,HIGH,LOW,HIGH,LOW,HP
20,HIGH,HIGH,LOW,HIGH,LOW,HP
21,HIGH,HIGH,HIGH,HIGH,LOW,VP
22,LOW,LOW,HIGH,LOW,LOW,VP
23,LOW,LOW,HIGH,LOW,LOW,VP
24,LOW,HIGH,LOW,LOW,LOW,VP
25,HIGH,LOW,HIGH,LOW,LOW,FR
26,HIGH,LOW,HIGH,LOW,LOW,VP
27,HIGH,HIGH,HIGH,HIGH,LOW,FR
28,HIGH,HIGH,HIGH,HIGH,LOW,DN
29,HIGH,HIGH,HIGH,LOW,LOW,FR
30,HIGH,HIGH,HIGH,HIGH,LOW,HP

APPENDIX J: Flowchart of Table Operator-Key Pair Selection



For Data Placement Algorithm (DPA) see Figure 10.

APPENDIX K: Flowchart of Table Partitioning Key Selection



For Data Placement Algorithm (DPA) see Figure 10.